Argonne NATIONAL LABORATORY

An approach to discrete adjoints for MPI-parallelized C++ models applied to the NASA/JPL Ice Sheet System Model

J. Utke / E. Larour

Argonne National Laboratory
NASA Jet Propulsion Laboratory

Feb/2014 TU Darmstadt, Germany

outline

- what this is for
- principles of AD
- changes in ISSM
- changes in Adol-C
- external solvers
- adjoinable MPI
- performance

Greenland

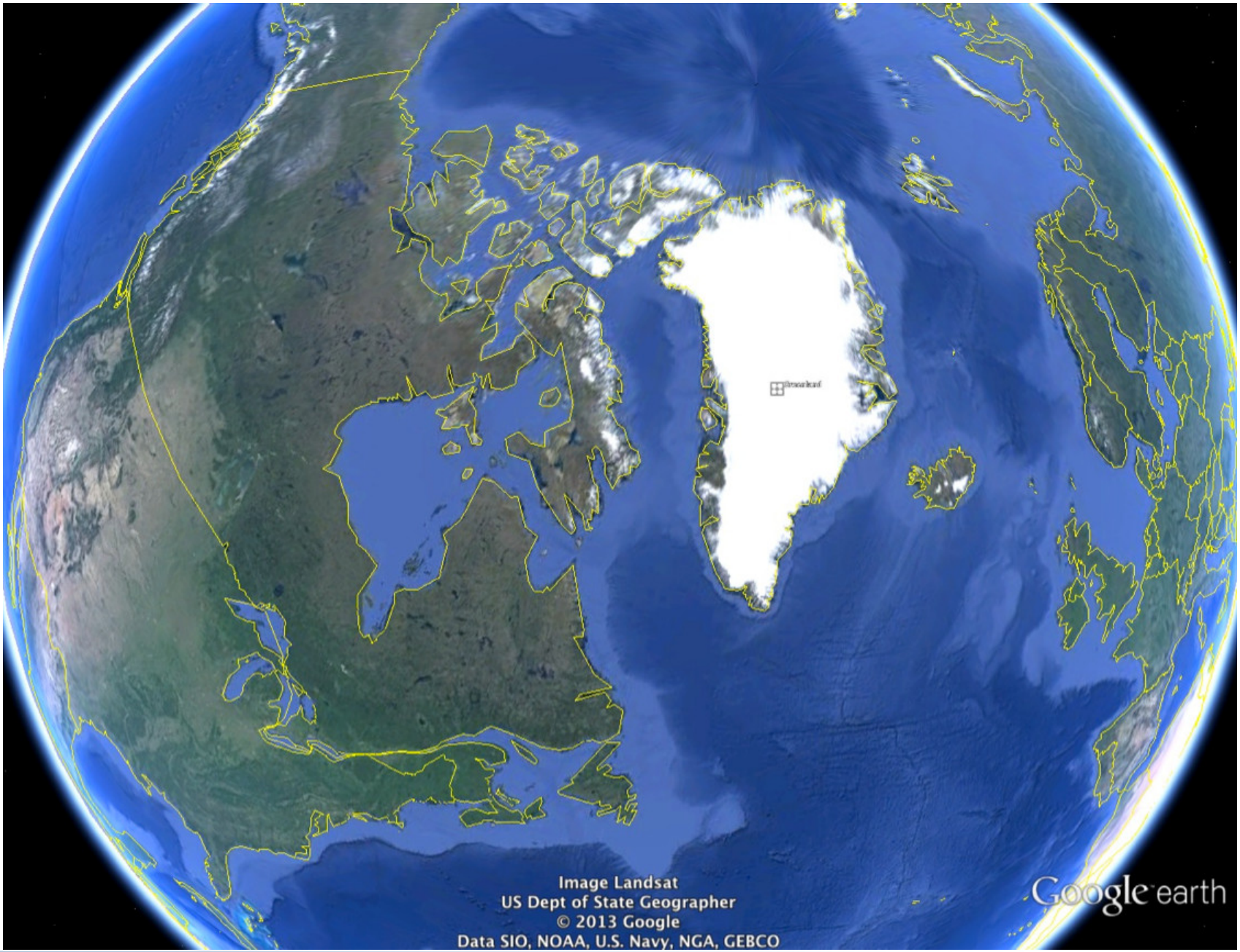
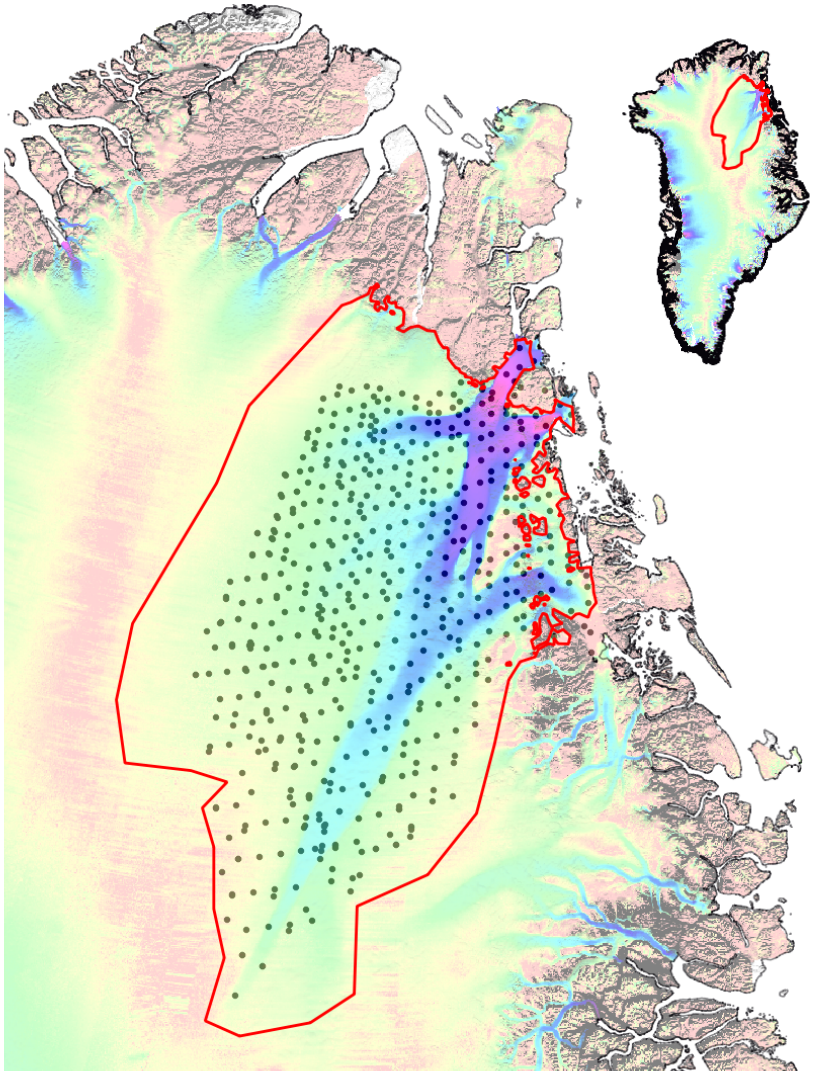


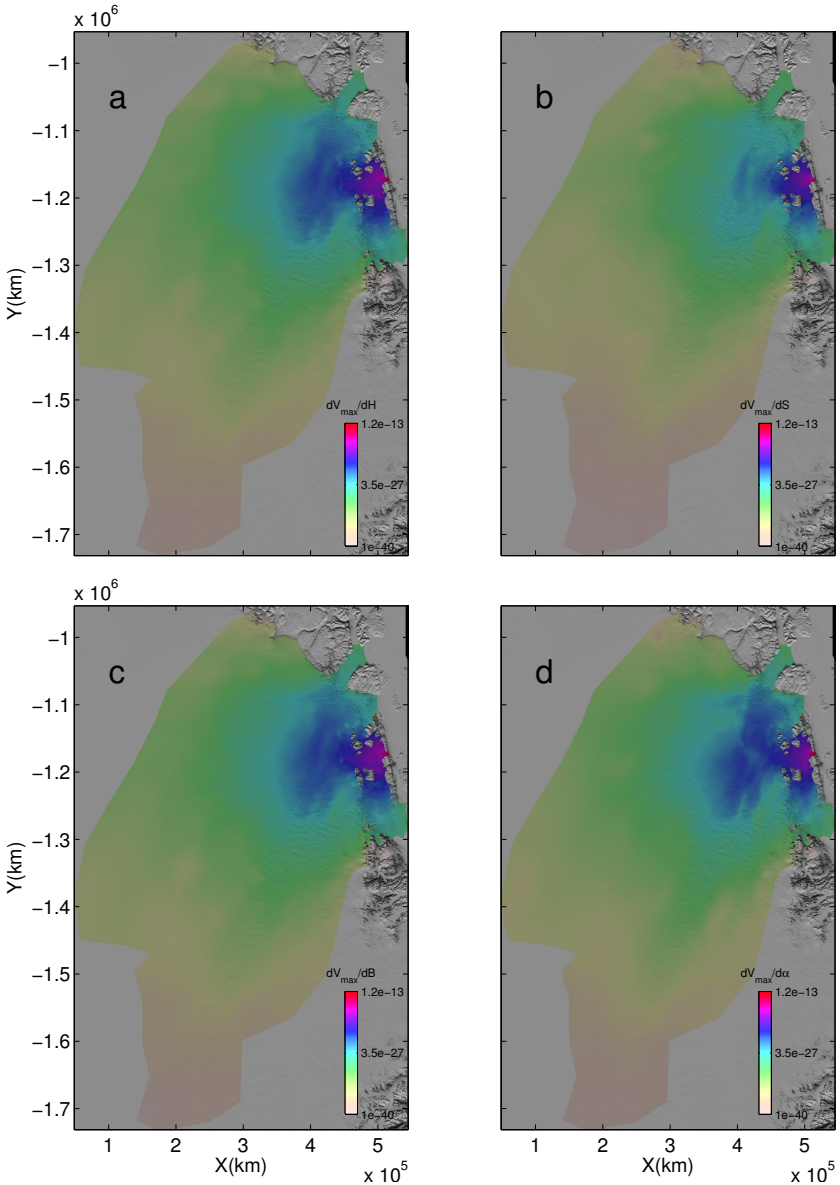
Image Landsat
US Dept of State Geographer
© 2013 Google
Data SIO, NOAA, U.S. Navy, NGA, GEBCO
Google earth

the North-Eastern Ice Stream on Greenland



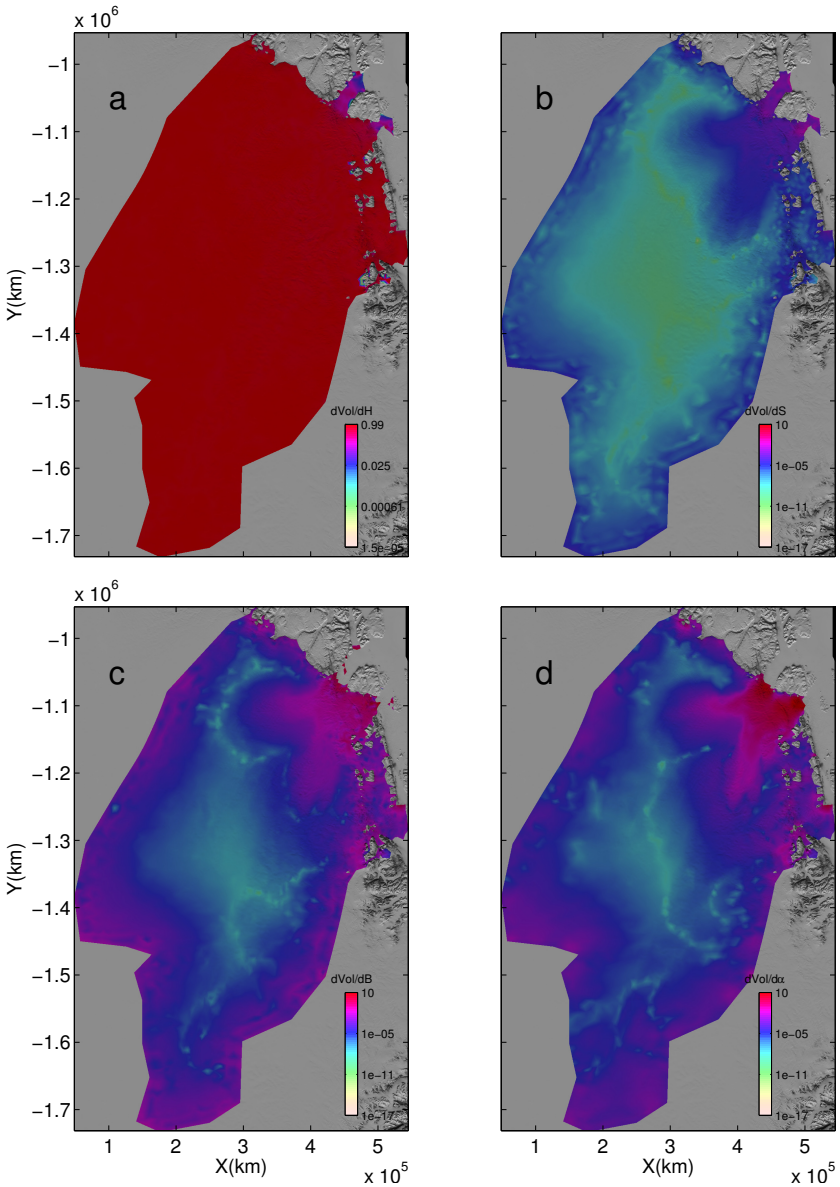
- velocity field
- red boundary shows domain of interest
- dots indicate observation data
- surface observations by satellite/stations
- drilling holes is expensive
- goal is model tuning for prediction of sea level rise

sensitivity studies - maximal velocity with respect to



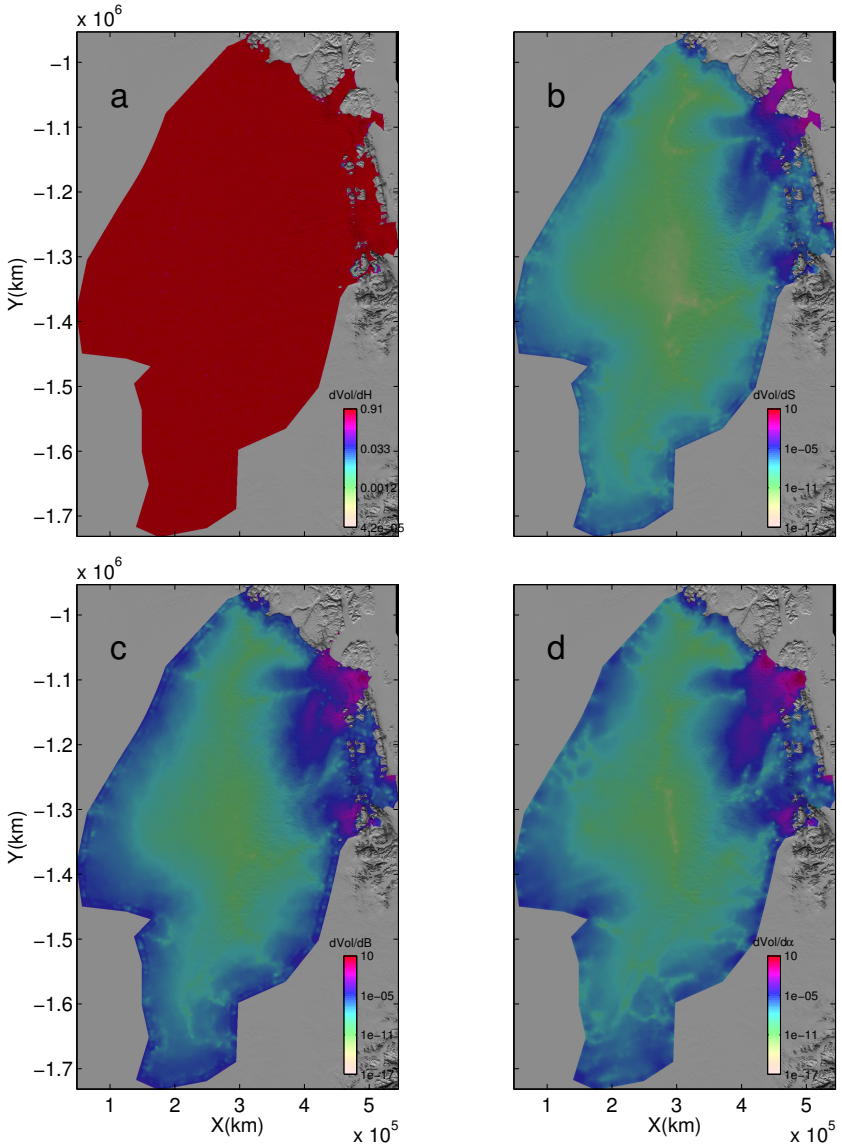
- a: H - ice thickness
- b: S - surface elevation
- c: B - bed elevation
- d: α - friction coefficient

sensitivity studies - volume with respect to



- a: H - ice thickness
- b: S - surface elevation
- c: B - bed elevation
- d: α - friction coefficient

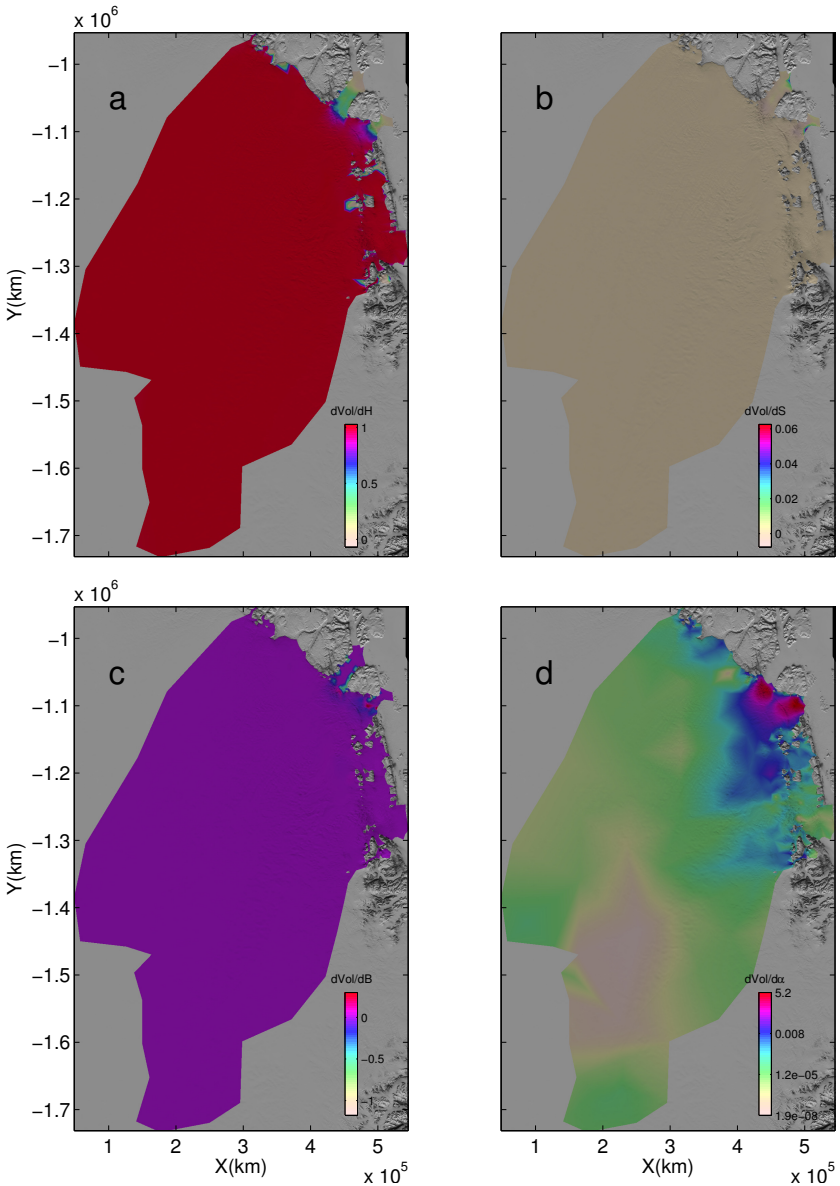
sensitivity studies (last week) - volume with respect to



- a: H - ice thickness
- b: S - surface elevation
- c: B - bed elevation
- d: α - friction coefficient

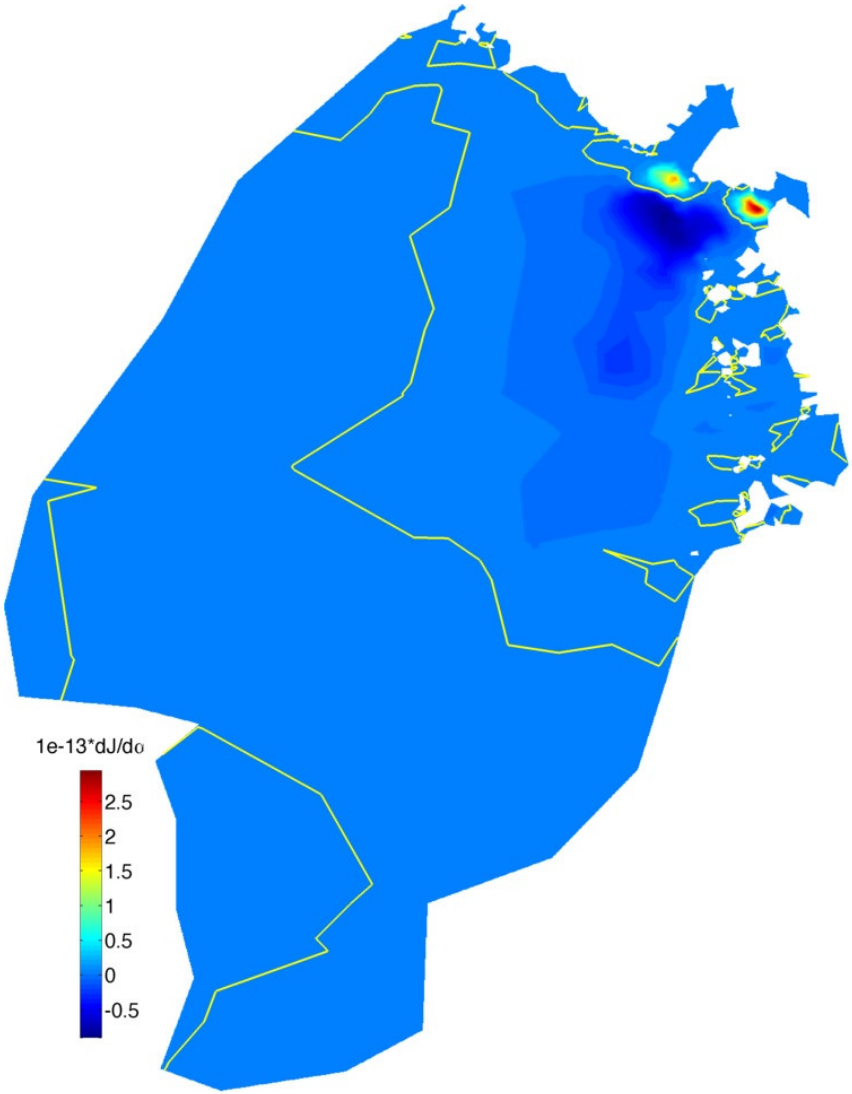
compared to earlier studies ran in higher resolution on Pleiades for longer model time

sensitivity studies - volume above floatation with respect to



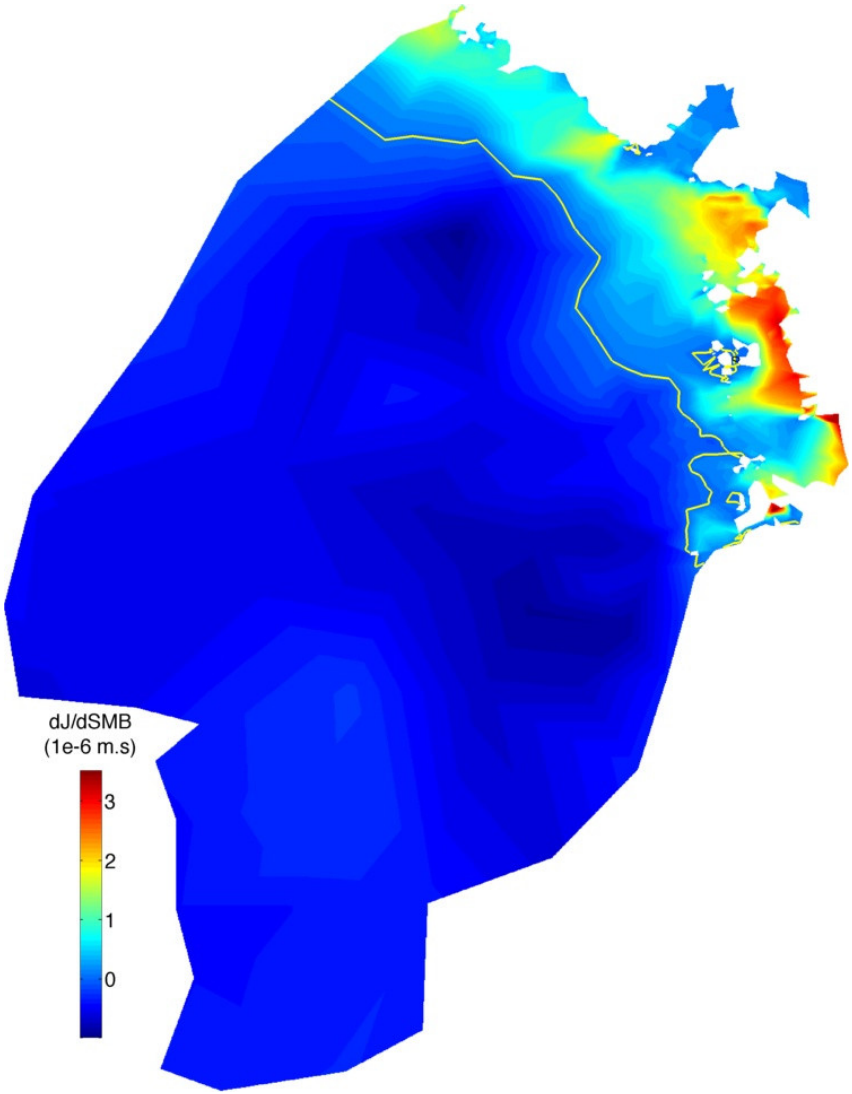
- a: H - ice thickness
- b: S - surface elevation
- c: B - bed elevation
- d: α - friction coefficient

model-to-observation misfit of S to internal state



- with respect to friction coefficients
- L^2 integrated over time
- yellow lines indicate gradient sign switch
- part of an gradient $\rightarrow$ line search optimization loop

model-to-observation misfit of S to external boundary



- with respect to snow-mass-balance
- snow fall given as (external) reanalysis of climate model runs
- hints at less snow fall on the coast, more inland
- means to adapt reanalysis if one assumes the ice sheet model is "correct"

presented at AGU meeting

why algorithmic differentiation?

given: some numerical model $\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x}) : \mathbb{R}^n \mapsto \mathbb{R}^m$
implemented as a (large / volatile) program

wanted: sensitivity analysis, optimization, parameter (state) estimation, higher-order approximation...

1. don't pretend we know nothing about the program
   (and take finite differences of an oracle)
2. get machine precision derivatives as $\boldsymbol{J}\dot{\boldsymbol{x}}$ or $\bar{\boldsymbol{y}}^T\boldsymbol{J}$ or ...
   (avoid approximation-versus-roundoff problem)
3. the reverse (aka adjoint) mode yields "cheap" gradients
4. if the program is large, so is the adjoint program, and
   so is the effort to do it manually ... easy to get wrong but hard to debug

$\Rightarrow$ use tools to do it **automatically!**

why algorithmic differentiation?

given: some numerical model $\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x}) : \mathbb{R}^n \mapsto \mathbb{R}^m$
implemented as a (large / volatile) program

wanted: sensitivity analysis, optimization, parameter (state) estimation, higher-order approximation...

1. don't pretend we know nothing about the program (and take finite differences of an oracle)
2. get machine precision derivatives as $\boldsymbol{J}\dot{\boldsymbol{x}}$ or $\bar{\boldsymbol{y}}^T\boldsymbol{J}$ or ... (avoid approximation-versus-roundoff problem)
3. the reverse (aka adjoint) mode yields "cheap" gradients
4. if the program is large, so is the adjoint program, and so is the effort to do it manually ... easy to get wrong but hard to debug

$\Rightarrow$ use tools to do it **automatically?**

why algorithmic differentiation?

given: some numerical model $\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x}) : \mathbb{R}^n \mapsto \mathbb{R}^m$
implemented as a (large / volatile) program

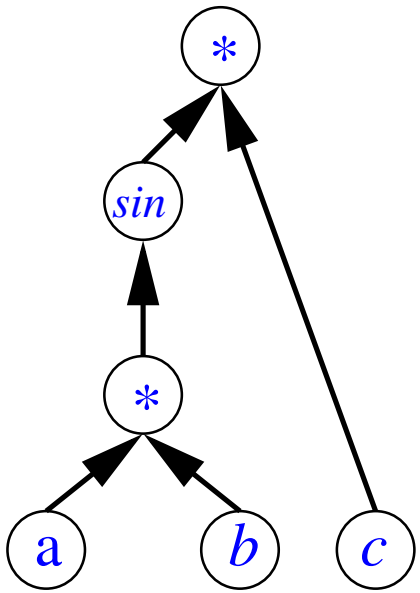
wanted: sensitivity analysis, optimization, parameter (state) estimation, higher-order approximation...

1. don't pretend we know nothing about the program
   (and take finite differences of an oracle)
2. get machine precision derivatives as $\boldsymbol{J}\dot{\boldsymbol{x}}$ or $\bar{\boldsymbol{y}}^T\boldsymbol{J}$ or ...
   (avoid approximation-versus-roundoff problem)
3. the reverse (aka adjoint) mode yields "cheap" gradients
4. if the program is large, so is the adjoint program, and
   so is the effort to do it manually ... easy to get wrong but hard to debug

$\Rightarrow$ use tools to do it at least semi-automatically!

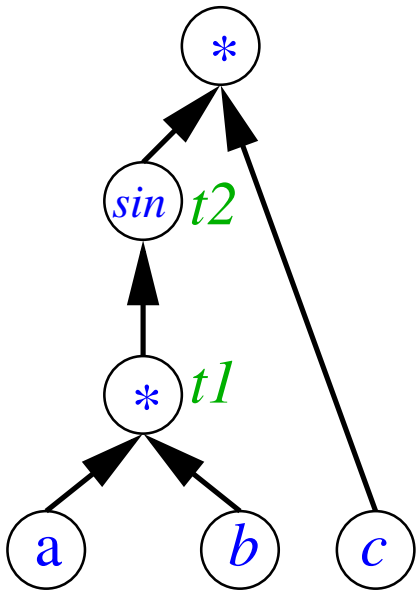
how does AD compute derivatives?

$f : y = sin(a * b) * c : \mathbb{R}^3 \mapsto \mathbb{R}$
yields a graph representing the order of computation:



how does AD compute derivatives?

$f : y = sin(a * b) * c : \mathbb{R}^3 \mapsto \mathbb{R}$
yields a graph representing the order of computation:



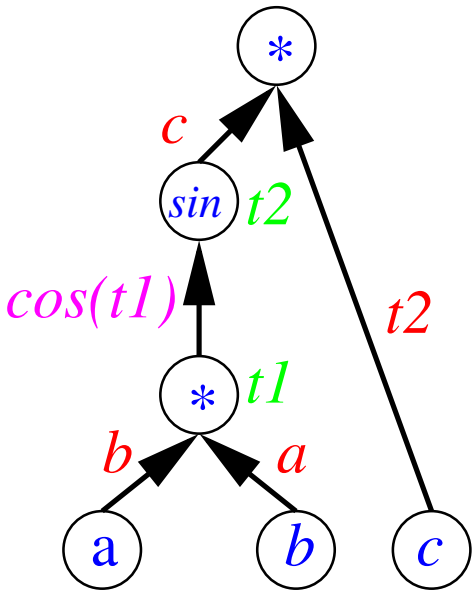
- *code list*$\rightarrow$ intermediate values $t1$ and $t2$

```
t1 = a*b

t2 = sin(t1)
 y = t2*c
```

how does AD compute derivatives?

$f : y = sin(a * b) * c : \mathbb{R}^3 \mapsto \mathbb{R}$
yields a graph representing the order of computation:

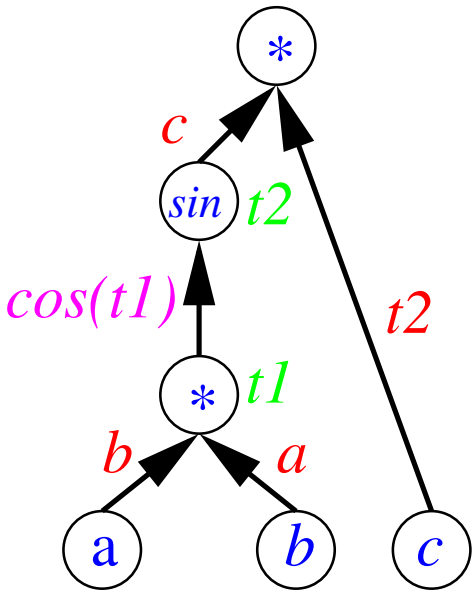


- *code list*$\rightarrow$ intermediate values $t1$ and $t2$
- each intrinsic $v = \phi(w, u)$ has local partials $\frac{\partial \phi}{\partial w}$, $\frac{\partial \phi}{\partial u}$
- e.g. `sin(t1)` yields `p1=cos(t1)`
- in our example all others are already stored in variables

```
t1 = a*b
p1 = cos(t1)
t2 = sin(t1)
 y = t2*c
```

how does AD compute derivatives?

$f : y = sin(a * b) * c : \mathbb{R}^3 \mapsto \mathbb{R}$
yields a graph representing the order of computation:



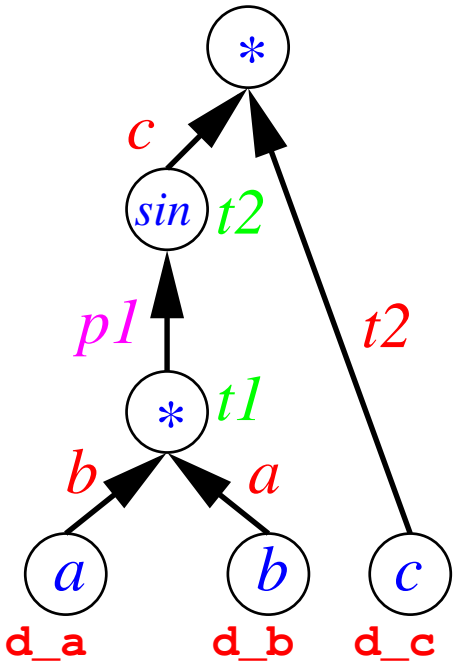
- *code list*$\rightarrow$ intermediate values $t1$ and $t2$
- each intrinsic $v = \phi(w, u)$ has local partials $\frac{\partial \phi}{\partial w}$, $\frac{\partial \phi}{\partial u}$
- e.g. `sin(t1)` yields `p1=cos(t1)`
- in our example all others are already stored in variables

```
t1 = a*b
p1 = cos(t1)
t2 = sin(t1)
 y = t2*c
```

What do we do with this?

forward mode with directional derivatives

- **associate** each variable v with a derivative $\dot{v}$
- take a point (a_0, b_0, c_0) and a direction $(\dot{a}, \dot{b}, \dot{c})$
- for each $v = \phi(w, u)$ propagate forward in order
  $\dot{v} = \frac{\partial \phi}{\partial w} \dot{w} + \frac{\partial \phi}{\partial u} \dot{u}$



- in practice: associate *by name* `[a,d_a]`
  or *by address* `[a%v,a%d]`
- interleave propagation computations

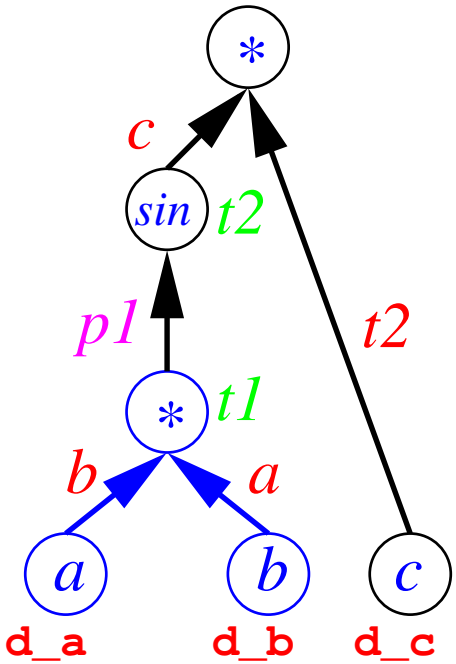
```
t1 = a*b

p1 = cos(t1)
t2 = sin(t1)

y = t2*c
```

forward mode with directional derivatives

- **associate** each variable v with a derivative $\dot{v}$
- take a point (a_0, b_0, c_0) and a direction $(\dot{a}, \dot{b}, \dot{c})$
- for each $v = \phi(w, u)$ propagate forward in order
  $\dot{v} = \frac{\partial \phi}{\partial w} \dot{w} + \frac{\partial \phi}{\partial u} \dot{u}$



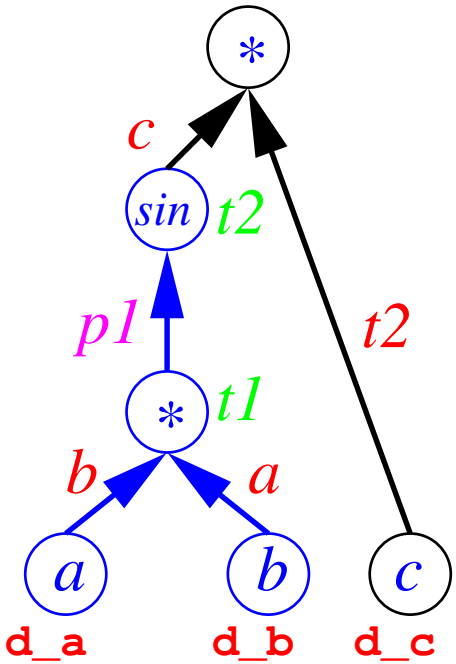
- in practice: associate *by name* `[a,d_a]`
  or *by address* `[a%v,a%d]`
- interleave propagation computations

```
t1 = a*b
d_t1 = d_a*b + d_b*a
p1 = cos(t1)
t2 = sin(t1)

y = t2*c
```

forward mode with directional derivatives

- **associate** each variable v with a derivative $\dot{v}$
- take a point (a_0, b_0, c_0) and a direction $(\dot{a}, \dot{b}, \dot{c})$
- for each $v = \phi(w, u)$ propagate forward in order
  $\dot{v} = \frac{\partial \phi}{\partial w} \dot{w} + \frac{\partial \phi}{\partial u} \dot{u}$

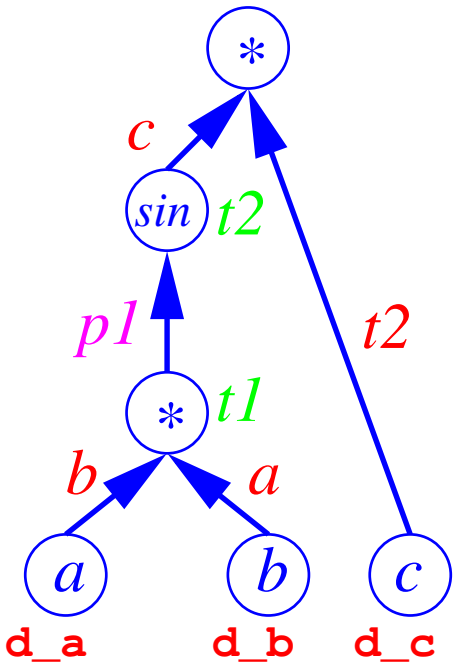


- in practice: associate *by name* `[a,d_a]`
  or *by address* `[a%v,a%d]`
- interleave propagation computations

```
t1 = a*b
d_t1 = d_a*b + d_b*a
p1 = cos(t1)
t2 = sin(t1)
d_t2 = d_t1*p1
y = t2*c
```

forward mode with directional derivatives

- **associate** each variable v with a derivative $\dot{v}$
- take a point (a_0, b_0, c_0) and a direction $(\dot{a}, \dot{b}, \dot{c})$
- for each $v = \phi(w, u)$ propagate forward in order $\dot{v} = \frac{\partial \phi}{\partial w} \dot{w} + \frac{\partial \phi}{\partial u} \dot{u}$

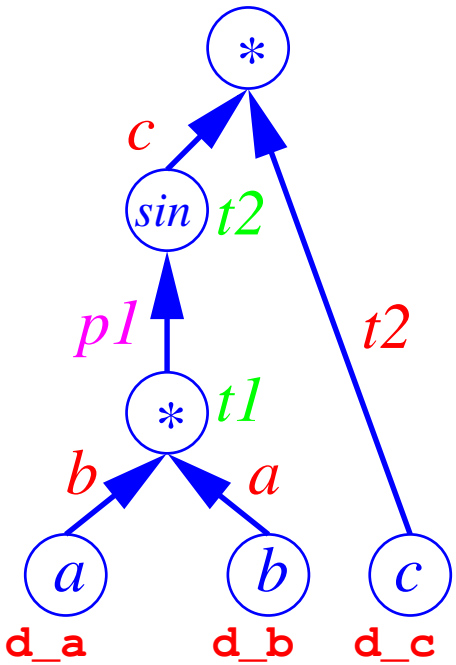


- in practice: associate *by name* `[a,d_a]` or *by address* `[a%v,a%d]`
- interleave propagation computations

```
t1 = a*b
d_t1 = d_a*b + d_b*a
p1 = cos(t1)
t2 = sin(t1)
d_t2 = d_t1*p1
y = t2*c
d_y = d_t2*c + d_c*t2
```

forward mode with directional derivatives

- **associate** each variable v with a derivative $\dot{v}$
- take a point (a_0, b_0, c_0) and a direction $(\dot{a}, \dot{b}, \dot{c})$
- for each $v = \phi(w, u)$ propagate forward in order
  $\dot{v} = \frac{\partial \phi}{\partial w} \dot{w} + \frac{\partial \phi}{\partial u} \dot{u}$



- in practice: associate *by name* `[a,d_a]`
  or *by address* `[a%v,a%d]`
- interleave propagation computations

```
t1 = a*b
d_t1 = d_a*b + d_b*a
p1 = cos(t1)
t2 = sin(t1)
d_t2 = d_t1*p1
y = t2*c
d_y = d_t2*c + d_c*t2
```

What is in d_y ?

d_y contains a projection

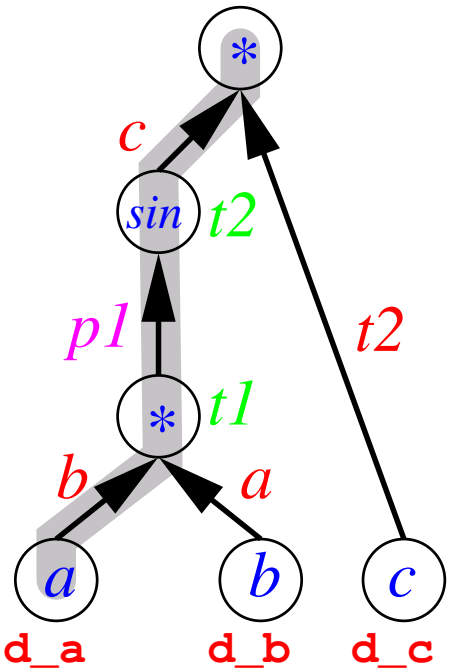
- $\dot{\boldsymbol{y}} = \boldsymbol{J}\dot{\boldsymbol{x}}$ computed at $\boldsymbol{x}_0$

d_y contains a projection

- $\dot{\boldsymbol{y}} = \boldsymbol{J}\dot{\boldsymbol{x}}$ computed at $\boldsymbol{x}_0$
- for example for $(\dot{a}, \dot{b}, \dot{c}) = (1, 0, 0)$

d_y contains a projection

- $\dot{\boldsymbol{y}} = \boldsymbol{J}\dot{\boldsymbol{x}}$ computed at $\boldsymbol{x}_0$
- for example for $(\dot{a}, \dot{b}, \dot{c}) = (1, 0, 0)$



- yields the first element of the gradient
- all gradient elements cost $\mathcal{O}(n)$ function evaluations

sidebar: simple overloaded operators for a*b

in C++:

```
struct Afloat{float v; float d;};

Afloat operator *(Afloat a, Afloat b) {
  Afloat r; int i;
  r.v=a.v*b.v; // value
  r.d=a.d*b.v+a.v*b.d; // derivative
  return r;
};

// other argument combinations
```

in Fortran:

```
module ATypes
 public :: Areal
 type Areal
   sequence
   real :: v,d
 end type
end module ATypes

module Amult
  use ATypes
  interface operator(*)
    module procedure multArealAreal
    ! other argument combinations
  end interface
contains
  function multArealAreal(a,b) result(r)
    type(Areal),intent(in)::a,b
    type(Areal)::r
    r%v=a%v*b%v ! value
    r%d=a%d*b%v+a%v*b%v ! derivative
  end function multArealAreal
end module Amult
```

sidebar: simple overloaded operators for a*b

in C++:

```
struct Afloat{float v; float d;};

Afloat operator *(Afloat a, Afloat b) {
  Afloat r; int i;
  r.v=a.v*b.v; // value
  r.d=a.d*b.v+a.v*b.d; // derivative
  return r;
};

// other argument combinations
```

in Fortran:

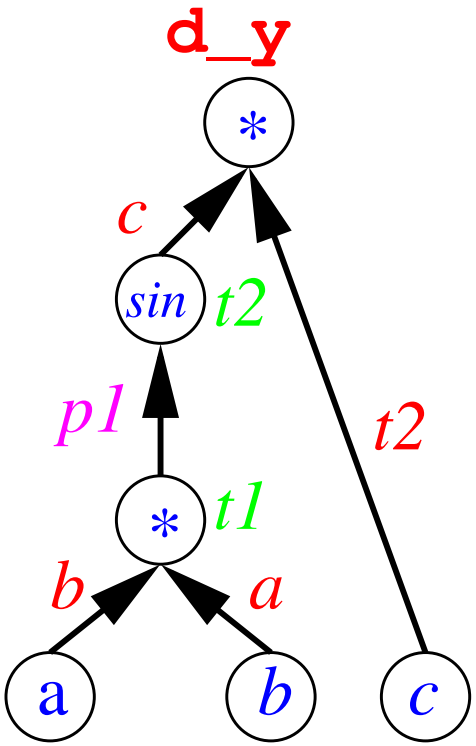
```
module ATypes
 public :: Areal
 type Areal
   sequence
   real :: v,d
 end type
end module ATypes

module Amult
  use ATypes
  interface operator(*)
    module procedure multArealAreal
    ! other argument combinations
  end interface
contains
  function multArealAreal(a,b) result(r)
    type(Areal),intent(in)::a,b
    type(Areal)::r
    r%v=a%v*b%v ! value
    r%d=a%d*b%v+a%v*b%v ! derivative
  end function multArealAreal
end module Amult
```

Operator Overloading $\Rightarrow$
A simple, relatively unintrusive way to augment semantics via a type change!

reverse mode with adjoints

- same association model
- take a point (a_0, b_0, c_0), compute y, pick a weight $\bar{y}$
- for each $v = \phi(w, u)$ propagate backward
  $\bar{w} += \frac{\partial \phi}{\partial w}\ \bar{v}; \quad \bar{u} += \frac{\partial \phi}{\partial u}\ \bar{v}; \quad \bar{v} = 0$

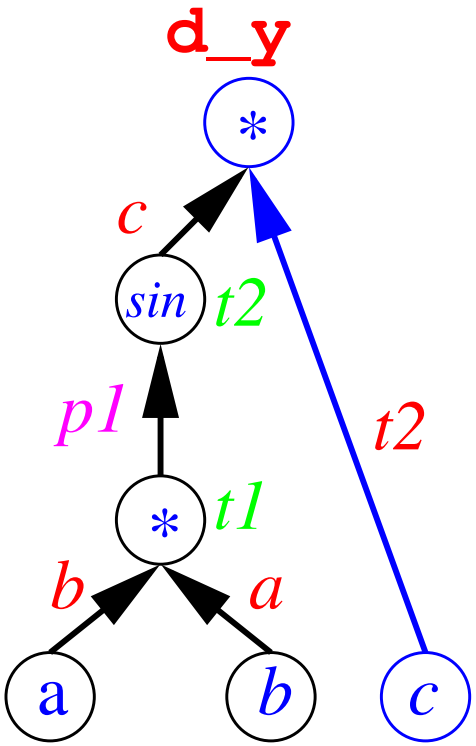


backward propagation code appended:

```
t1 = a*b
p1 = cos(t1)
t2 = sin(t1)
y = t2*c
```

reverse mode with adjoints

- same association model
- take a point (a_0, b_0, c_0), compute y, pick a weight $\bar{y}$
- for each $v = \phi(w, u)$ propagate backward
  $\bar{w} += \frac{\partial \phi}{\partial w} \; \bar{v}; \quad \bar{u} += \frac{\partial \phi}{\partial u} \; \bar{v}; \quad \bar{v} = 0$

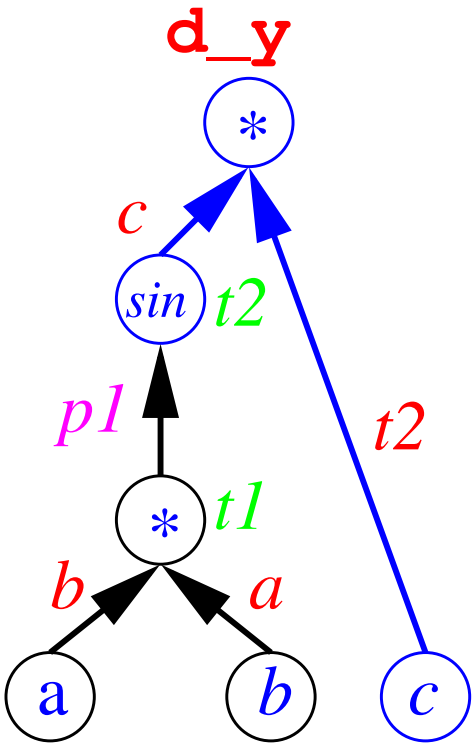


backward propagation code appended:

```
t1 = a*b
p1 = cos(t1)
t2 = sin(t1)
y = t2*c
d_c = t2*d_y
```

reverse mode with adjoints

- same association model
- take a point (a_0, b_0, c_0), compute y, pick a weight $\bar{y}$
- for each $v = \phi(w, u)$ propagate backward
  $\bar{w} += \frac{\partial \phi}{\partial w} \; \bar{v}; \quad \bar{u} += \frac{\partial \phi}{\partial u} \; \bar{v}; \quad \bar{v} = 0$

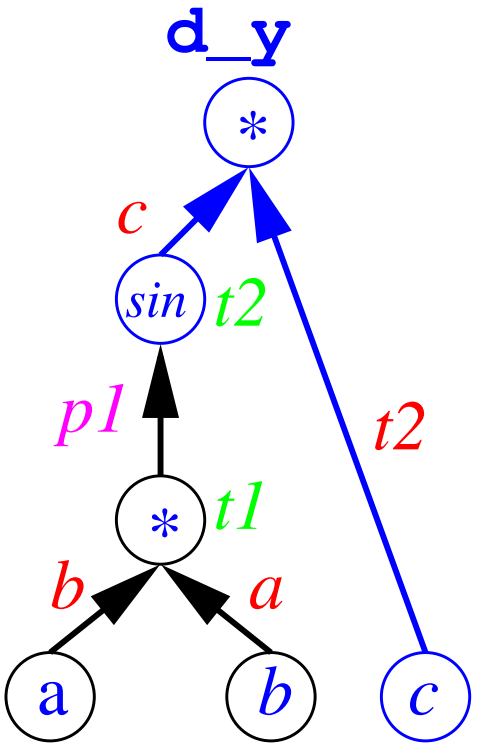


backward propagation code appended:

```
t1 = a*b
p1 = cos(t1)
t2 = sin(t1)
y = t2*c
d_c = t2*d_y
d_t2 = c*d_y
```

reverse mode with adjoints

- same association model
- take a point (a_0, b_0, c_0), compute y, pick a weight $\bar{y}$
- for each $v = \phi(w, u)$ propagate backward
  $\bar{w} += \frac{\partial \phi}{\partial w}\ \bar{v}; \quad \bar{u} += \frac{\partial \phi}{\partial u}\ \bar{v}; \quad \bar{v} = 0$

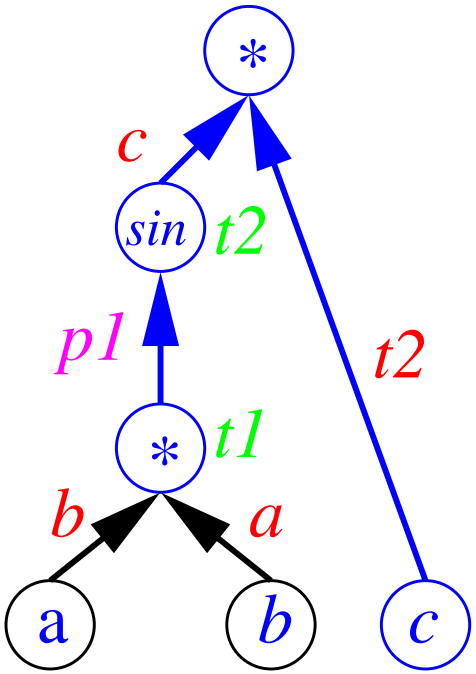


backward propagation code appended:

```
t1 = a*b
p1 = cos(t1)
t2 = sin(t1)
y = t2*c
d_c = t2*d_y
d_t2 = c*d_y
d_y = 0
```

reverse mode with adjoints

- same association model
- take a point (a_0, b_0, c_0), compute y, pick a weight $\bar{y}$
- for each $v = \phi(w, u)$ propagate backward
  $\bar{w} += \frac{\partial \phi}{\partial w} \; \bar{v}; \quad \bar{u} += \frac{\partial \phi}{\partial u} \; \bar{v}; \quad \bar{v} = 0$

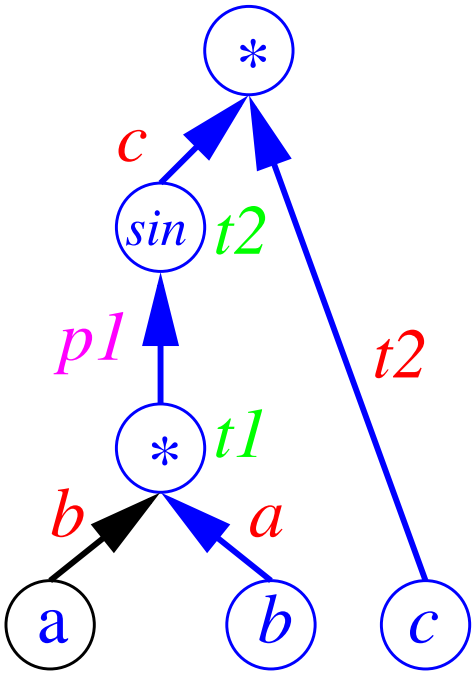


backward propagation code appended:

```
t1 = a*b
p1 = cos(t1)
t2 = sin(t1)
y = t2*c
d_c = t2*d_y
d_t2 = c*d_y
d_y = 0
d_t1 = p1*d_t2
```

reverse mode with adjoints

- same association model
- take a point (a_0, b_0, c_0), compute y, pick a weight $\bar{y}$
- for each $v = \phi(w, u)$ propagate backward
  $\bar{w} += \frac{\partial \phi}{\partial w} \; \bar{v}; \quad \bar{u} += \frac{\partial \phi}{\partial u} \; \bar{v}; \quad \bar{v} = 0$

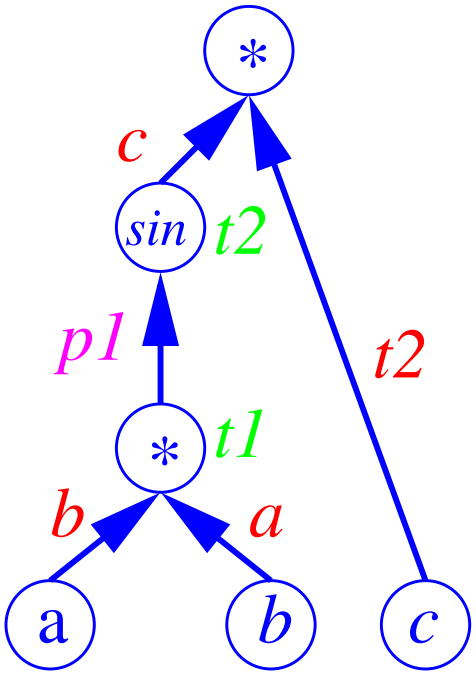


backward propagation code appended:

```
t1 = a*b
p1 = cos(t1)
t2 = sin(t1)
y = t2*c
d_c = t2*d_y
d_t2 = c*d_y
d_y = 0
d_t1 = p1*d_t2
d_b = a*d_t1
```

reverse mode with adjoints

- same association model
- take a point (a_0, b_0, c_0), compute y, pick a weight $\bar{y}$
- for each $v = \phi(w, u)$ propagate backward
  $\bar{w} += \frac{\partial \phi}{\partial w} \; \bar{v}; \quad \bar{u} += \frac{\partial \phi}{\partial u} \; \bar{v}; \quad \bar{v} = 0$

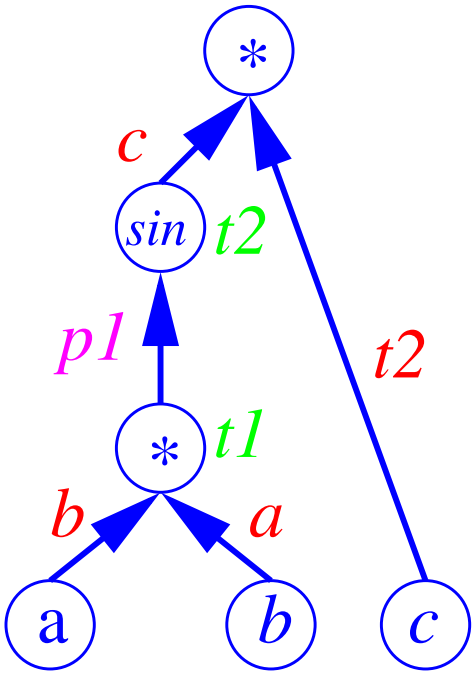


backward propagation code appended:

```
t1 = a*b
p1 = cos(t1)
t2 = sin(t1)
y = t2*c
d_c = t2*d_y
d_t2 = c*d_y
d_y = 0
d_t1 = p1*d_t2
d_b = a*d_t1
d_a = b*d_t1
```

reverse mode with adjoints

- same association model
- take a point (a_0, b_0, c_0), compute y, pick a weight $\bar{y}$
- for each $v = \phi(w, u)$ propagate backward
  $\bar{w} += \frac{\partial \phi}{\partial w} \; \bar{v}; \quad \bar{u} += \frac{\partial \phi}{\partial u} \; \bar{v}; \quad \bar{v} = 0$



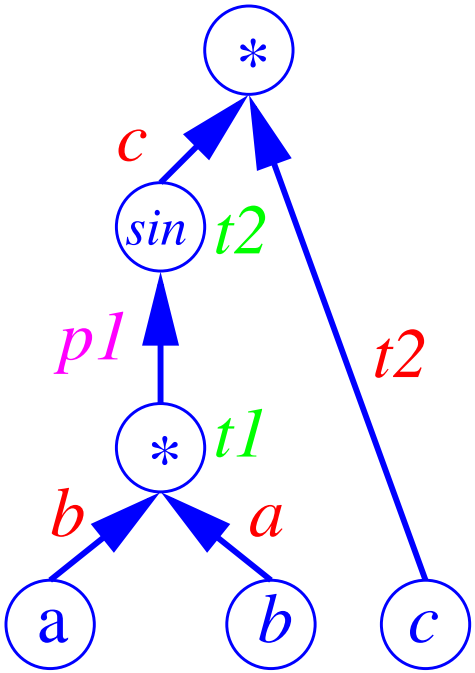
backward propagation code appended:

```
t1 = a*b
p1 = cos(t1)
t2 = sin(t1)
y = t2*c
d_c = t2*d_y
d_t2 = c*d_y
d_y = 0
d_t1 = p1*d_t2
d_b = a*d_t1
d_a = b*d_t1
```

What is in (d_a,d_b,d_c)?

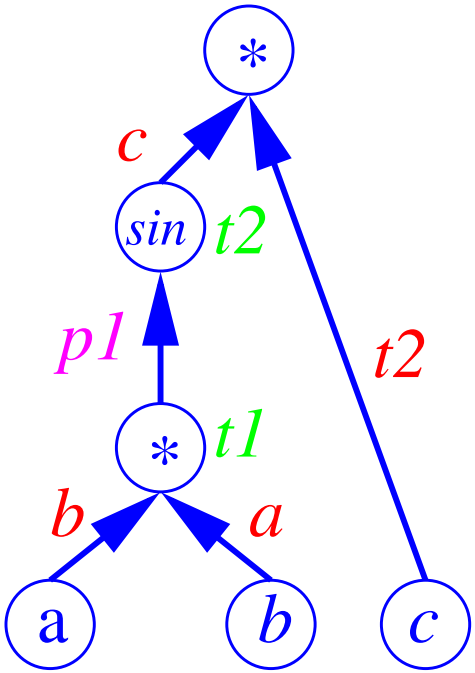
(d_a,d_b,d_c) contains a projection

- $\bar{\boldsymbol{x}} = \bar{\boldsymbol{y}}^T \boldsymbol{J}$ computed at $\boldsymbol{x}_0$



(d_a,d_b,d_c) contains a projection

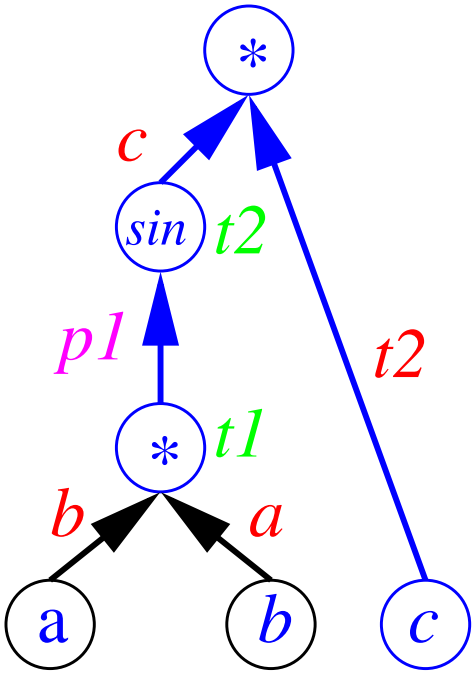
- $\bar{\boldsymbol{x}} = \bar{\boldsymbol{y}}^T \boldsymbol{J}$ computed at $\boldsymbol{x}_0$
- for example for $\bar{y} = 1$ we have $[\bar{a}, \bar{b}, \bar{c}] = \nabla f$



- all gradient elements cost $\mathcal{O}(1)$ function evaluations

(d_a,d_b,d_c) contains a projection

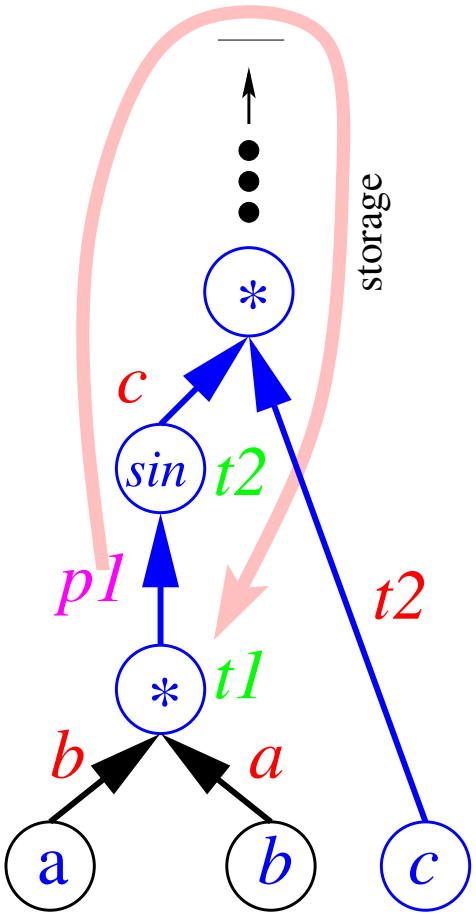
- $\bar{\boldsymbol{x}} = \bar{\boldsymbol{y}}^T \boldsymbol{J}$ computed at $\boldsymbol{x}_0$
- for example for $\bar{y} = 1$ we have $[\bar{a}, \bar{b}, \bar{c}] = \nabla f$



- all gradient elements cost $\mathcal{O}(1)$ function evaluations
- but consider when $p1$ is computed and when it is used

(d_a,d_b,d_c) contains a projection

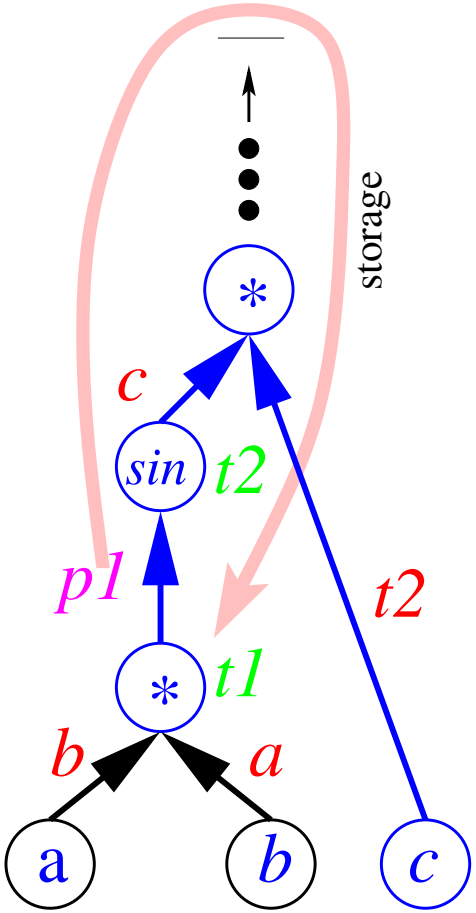
- $\bar{\boldsymbol{x}} = \bar{\boldsymbol{y}}^T \boldsymbol{J}$ computed at $\boldsymbol{x}_0$
- for example for $\bar{y} = 1$ we have $[\bar{a}, \bar{b}, \bar{c}] = \nabla f$



- all gradient elements cost $\mathcal{O}(1)$ function evaluations
- but consider when $p1$ is computed and when it is used
- storage requirements grow with the length of the computation
- typically mitigated by recomputation from checkpoints

(d_a, d_b, d_c) contains a projection

- $\bar{\boldsymbol{x}} = \bar{\boldsymbol{y}}^T \boldsymbol{J}$ computed at $\boldsymbol{x}_0$
- for example for $\bar{y} = 1$ we have $[\bar{a}, \bar{b}, \bar{c}] = \nabla f$



- all gradient elements cost $\mathcal{O}(1)$ function evaluations
- but consider when $p1$ is computed and when it is used
- storage requirements grow with the length of the computation
- typically mitigated by recomputation from checkpoints

Reverse mode with Adol-C.

Adol-C example

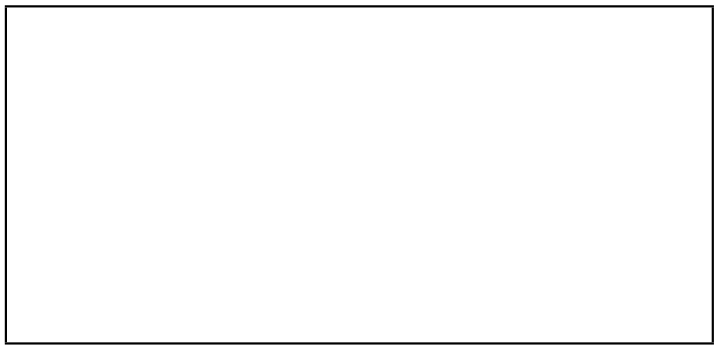
Adol-C: open source C++ operator-overloading library

Speelpenning example: $y = \prod_i x_i$ evaluated at $x_i = \frac{i+1}{i+2}$

```
 double *x = new  double[n];
 double  t = 1;
 double y;

for(i=0; i<n; i++) {
  x[i]   = (i+1.0)/(i+2.0);
  t *= x[i]; }
y = t;

delete[] x;
```

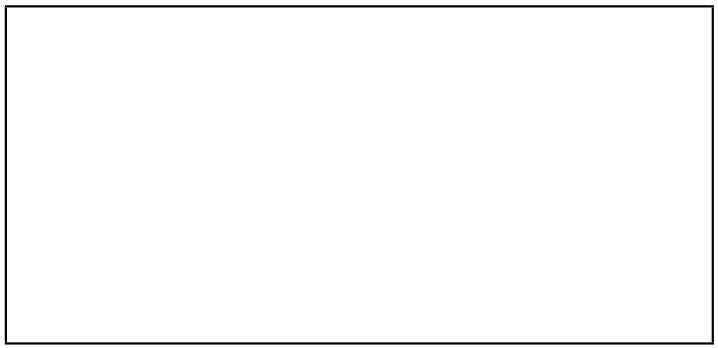


Adol-C example

Adol-C: open source C++ operator-overloading library

Speelpenning example: $y = \prod_i x_i$ evaluated at $x_i = \frac{i+1}{i+2}$

```
#include "adolc.h"
adouble *x = new adouble[n];
adouble  t = 1;
 double y;
trace_on(1);
for(i=0; i<n; i++) {
  x[i] <<= (i+1.0)/(i+2.0);
  t *= x[i]; }
t >>= y;
trace_off();
delete[] x;
```



Adol-C example

Adol-C: open source C++ operator-overloading library

Speelpenning example: $y = \prod_i x_i$ evaluated at $x_i = \frac{i+1}{i+2}$

```
#include "adolc.h"
adouble *x = new adouble[n];
adouble  t = 1;
 double y;
trace_on(1);
for(i=0; i<n; i++) {
  x[i] <<= (i+1.0)/(i+2.0);
  t *= x[i]; }
t >>= y;
trace_off();
delete[] x;
```

use a driver :

```
gradient(tag,
         n,
         x[n],
         g[n])
```

ISSM

- C++ model

ISSM

- C++ model
- $\approx$ 750 files, 100K lines, 10 developers

ISSM

- C++ model
- $\approx$ 750 files, 100K lines, 10 developers
- uses templates, virtual methods, multiple inheritance (i.e. isn't just glorified C)

ISSM

- C++ model
- $\approx$ 750 files, 100K lines, 10 developers
- uses templates, virtual methods, multiple inheritance (i.e. isn't just glorified C)
- only the model core is C++, data pre/post processing done with Matlab / Python

ISSM

- C++ model
- $\approx$ 750 files, 100K lines, 10 developers
- uses templates, virtual methods, multiple inheritance (i.e. isn't just glorified C)
- only the model core is C++, data pre/post processing done with Matlab / Python
- parallelized with MPI

ISSM

- C++ model
- $\approx$ 750 files, 100K lines, 10 developers
- uses templates, virtual methods, multiple inheritance (i.e. isn't just glorified C)
- only the model core is C++, data pre/post processing done with Matlab / Python
- parallelized with MPI
- runs on *nix; NASA's Pleiades $\leftrightarrow$ Android

ISSM

- C++ model
- $\approx$ 750 files, 100K lines, 10 developers
- uses templates, virtual methods, multiple inheritance (i.e. isn't just glorified C)
- only the model core is C++, data pre/post processing done with Matlab / Python
- parallelized with MPI
- runs on *nix; NASA's Pleiades $\leftrightarrow$ Android
- has extensive regression testing (incl. the numerical results)

ISSM

- C++ model
- $\approx$ 750 files, 100K lines, 10 developers
- uses templates, virtual methods, multiple inheritance (i.e. isn't just glorified C)
- only the model core is C++, data pre/post processing done with Matlab / Python
- parallelized with MPI
- runs on *nix; NASA's Pleiades $\leftrightarrow$ Android
- has extensive regression testing (incl. the numerical results)
- uses libraries (meshing, partitioning, solvers)

AdolC-ify ISSM (1)

- `typedef` an `IssmDouble` and an `IssmPDouble` and switch on and off via `_HAVE_ADOLC_` configure define

AdolC-ify ISSM (1)

- `typedef` an `IssmDouble` and an `IssmPDouble` and switch on and off via `_HAVE_ADOLC_` configure define
- contributors deliver code in terms of `doubles` and expert developer catogorizes those into `IssmDoubles` and `IssmPDoubles`
  $\Longrightarrow$ easy check

AdolC-ify ISSM (1)

- `typedef` an `IssmDouble` and an `IssmPDouble` and switch on and off via `_HAVE_ADOLC_` configure define
- contributors deliver code in terms of `doubles` and expert developer catogorizes those into `IssmDoubles` and `IssmPDoubles`
  $\Longrightarrow$ easy check
- replace all `mallocs`, `news` and `frees`, `deletes` by templated `xNew` / `xDelete` incl. variants for 2-D arrays
  $\Longrightarrow$ safer, cleaner, more efficient; easy check

AdolC-ify ISSM (1)

- `typedef` an `IssmDouble` and an `IssmPDouble` and switch on and off via `_HAVE_ADOLC_` configure define
- contributors deliver code in terms of `doubles` and expert developer catogorizes those into `IssmDoubles` and `IssmPDoubles`
  $\Longrightarrow$ easy check
- replace all `mallocs`, `news` and `frees`, `deletes` by templated `xNew` / `xDelete` incl. variants for 2-D arrays
  $\Longrightarrow$ safer, cleaner, more efficient; easy check
- templatize data containers (partially done)
- some undue activation (still) forced through Matlab interface

AdolC-ify ISSM (1)

- `typedef` an `IssmDouble` and an `IssmPDouble` and switch on and off via `_HAVE_ADOLC_` configure define
- contributors deliver code in terms of `doubles` and expert developer catogorizes those into `IssmDoubles` and `IssmPDoubles`
  $\Longrightarrow$ easy check
- replace all `mallocs`, `news` and `frees`, `deletes` by templated `xNew` / `xDelete` incl. variants for 2-D arrays
  $\Longrightarrow$ safer, cleaner, more efficient; easy check
- templatize data containers (partially done)
- some undue activation (still) forced through Matlab interface
- passing data to passive code with templated `reCast`

AdolC-ify ISSM (1)

- `typedef` an `IssmDouble` and an `IssmPDouble` and switch on and off via `_HAVE_ADOLC_` configure define
- contributors deliver code in terms of `doubles` and expert developer catogorizes those into `IssmDoubles` and `IssmPDoubles`
  $\Longrightarrow$ easy check
- replace all `mallocs`, `news` and `frees`, `deletes` by templated `xNew` / `xDelete` incl. variants for 2-D arrays
  $\Longrightarrow$ safer, cleaner, more efficient; easy check
- templatize data containers (partially done)
- some undue activation (still) forced through Matlab interface
- passing data to passive code with templated `reCast`
- `reCast` injections represent majority of the manual adaptation work

AdolC-ify ISSM (2)

- pick a simple(!) setup to start with and establish consistency with FD tests

AdolC-ify ISSM (2)

- pick a simple(!) setup to start with and establish consistency with FD tests
- few time steps, coarse resolution
- sequential, dense LU solve from GSL - needs wrapping

AdolC-ify ISSM (2)

- pick a simple(!) setup to start with and establish consistency with FD tests
- few time steps, coarse resolution
- sequential, dense LU solve from GSL - needs wrapping

AdolC-ify ISSM (2) & change Adol-C

- pick a simple(!) setup to start with and establish consistency with FD tests
- few time steps, coarse resolution
- sequential, dense LU solve from GSL - needs wrapping

sidebar: external solvers/frameworks

- interfaces implement *fixed* mathematical meaning
- may be a "black box" (different language, proprietary)

sidebar: external solvers/frameworks

- interfaces implement *fixed* mathematical meaning
- may be a "black box" (different language, proprietary)
- hopefully has derivatives easily implementable with the library calls, e.g. BLAS,
- linear solves $\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}$
  - one can show $\dot{\boldsymbol{x}} = \boldsymbol{A}^{-1}(\dot{\boldsymbol{b}} - \dot{\boldsymbol{A}}\boldsymbol{x})$
  - $\bar{\boldsymbol{b}} = \boldsymbol{A}^{-T}\bar{\boldsymbol{x}};\ \bar{\boldsymbol{A}} += -\bar{\boldsymbol{b}}\boldsymbol{x}^T$
- typically requires single call encapsulation (esp. for LU-style solvers)

sidebar: external solvers/frameworks

- interfaces implement *fixed* mathematical meaning
- may be a "black box" (different language, proprietary)
- hopefully has derivatives easily implementable with the library calls, e.g. BLAS,
- linear solves $\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}$
  - one can show $\dot{\boldsymbol{x}} = \boldsymbol{A}^{-1}(\dot{\boldsymbol{b}} - \dot{\boldsymbol{A}}\boldsymbol{x})$
  - $\bar{\boldsymbol{b}} = \boldsymbol{A}^{-T}\bar{\boldsymbol{x}};\ \bar{\boldsymbol{A}} += -\bar{\boldsymbol{b}}\boldsymbol{x}^T$
- typically requires single call encapsulation (esp. for LU-style solvers)
- done for ISSM with GNU Scientific Library (GSL) and MUMPS (MPI parallelized !)

sidebar: external solvers/frameworks

- interfaces implement *fixed* mathematical meaning
- may be a "black box" (different language, proprietary)
- hopefully has derivatives easily implementable with the library calls, e.g. BLAS,
- linear solves $\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}$
  - one can show $\dot{\boldsymbol{x}} = \boldsymbol{A}^{-1}(\dot{\boldsymbol{b}} - \dot{\boldsymbol{A}}\boldsymbol{x})$
  - $\bar{\boldsymbol{b}} = \boldsymbol{A}^{-T}\bar{\boldsymbol{x}};\ \bar{\boldsymbol{A}} += -\bar{\boldsymbol{b}}\boldsymbol{x}^T$
- typically requires single call encapsulation (esp. for LU-style solvers)
- done for ISSM with GNU Scientific Library (GSL) and MUMPS (MPI parallelized !)
- *always consider* augment convergence criterion for iterative numerical methods

sidebar: external solvers/frameworks

- interfaces implement *fixed* mathematical meaning
- may be a "black box" (different language, proprietary)
- hopefully has derivatives easily implementable with the library calls, e.g. BLAS,
- linear solves $\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}$
  - one can show $\dot{\boldsymbol{x}} = \boldsymbol{A}^{-1}(\dot{\boldsymbol{b}} - \dot{\boldsymbol{A}}\boldsymbol{x})$
  - $\bar{\boldsymbol{b}} = \boldsymbol{A}^{-T}\bar{\boldsymbol{x}};\ \bar{\boldsymbol{A}} += -\bar{\boldsymbol{b}}\boldsymbol{x}^T$
- typically requires single call encapsulation (esp. for LU-style solvers)
- done for ISSM with GNU Scientific Library (GSL) and MUMPS (MPI parallelized !)
- *always consider* augment convergence criterion for iterative numerical methods
- efficiency considerations, e.g. for fix point iterations $\boldsymbol{x}_{k+1} = \boldsymbol{f}(\boldsymbol{x}_k)$

AdolC-ify ISSM (2)

- pick a simple(!) setup to start with and establish consistency with FD tests
- few time steps, coarse resolution
- sequential, dense LU solve from GSL - needs wrapping

AdolC-ify ISSM (2)

- pick a simple(!) setup to start with and establish consistency with FD tests
- few time steps, coarse resolution
- sequential, dense LU solve from GSL - needs wrapping
- Adol-C has had an "external" interface for `func(x,y)` all along **but**:

AdolC-ify ISSM (2)

- pick a simple(!) setup to start with and establish consistency with FD tests
- few time steps, coarse resolution
- sequential, dense LU solve from GSL - needs wrapping
- Adol-C has had an "external" interface for `func(x,y)` all along **but**:
  - the forward variants passed only $\dot{x}$, $\dot{y}$ and not `x`, `y` themselves $\Longrightarrow$ added

AdolC-ify ISSM (2) & change Adol-C

- pick a simple(!) setup to start with and establish consistency with FD tests
- few time steps, coarse resolution
- sequential, dense LU solve from GSL - needs wrapping
- Adol-C has had an "external" interface for `func(x,y)` all along **but**:
  - the forward variants passed only $\dot{x}$, $\dot{y}$ and not `x`, `y` themselves $\Longrightarrow$ added
  - added sanity checks for consecutive locations

AdolC-ify ISSM (2) & change Adol-C

- pick a simple(!) setup to start with and establish consistency with FD tests
- few time steps, coarse resolution
- sequential, dense LU solve from GSL - needs wrapping
- Adol-C has had an "external" interface for `func(x,y)` all along **but**:
  - the forward variants passed only $\dot{x}$, $\dot{y}$ and not `x`, `y` themselves $\Longrightarrow$ added
  - added sanity checks for consecutive locations
  - the forward/reverse handlers taped/restored `x`, `y` values whether needed or not $\Longrightarrow$ added controls

AdolC-ify ISSM (2) & change Adol-C

- pick a simple(!) setup to start with and establish consistency with FD tests
- few time steps, coarse resolution
- sequential, dense LU solve from GSL - needs wrapping
- Adol-C has had an "external" interface for `func(x,y)` all along **but**:
  - the forward variants passed only $\dot{x}$, $\dot{y}$ and not `x`, `y` themselves $\Longrightarrow$ added
  - added sanity checks for consecutive locations
  - the forward/reverse handlers taped/restored `x`, `y` values whether needed or not $\Longrightarrow$ added controls
  - time stepping $\rightarrow$ multiple solves with adaptive meshing $\rightarrow$ changing system dimensions $\Longrightarrow$ added tracking for maximum dimensions for single allocation of reusable help buffers

AdolC-ify ISSM (2) & change Adol-C

- pick a simple(!) setup to start with and establish consistency with FD tests
- few time steps, coarse resolution
- sequential, dense LU solve from GSL - needs wrapping
- Adol-C has had an "external" interface for `func(x,y)` all along **but**:
  - the forward variants passed only $\dot{x}$, $\dot{y}$ and not `x`, `y` themselves $\Longrightarrow$ added
  - added sanity checks for consecutive locations
  - the forward/reverse handlers taped/restored `x`, `y` values whether needed or not $\Longrightarrow$ added controls
  - time stepping $\rightarrow$ multiple solves with adaptive meshing $\rightarrow$ changing system dimensions $\Longrightarrow$ added tracking for maximum dimensions for single allocation of reusable help buffers
- got first regression tests passing with matching forward & reverse derivatives

AdolC-ify ISSM (3)

- so far the triumph of the "mature" tool

AdolC-ify ISSM (3)

- so far the triumph of the "mature" tool
- but then we expanded test cases and ...
- first problem - wrong forward values computed:

AdolC-ify ISSM (3)

- so far the triumph of the "mature" tool
- but then we expanded test cases and ...
- first problem - wrong forward values computed:
  - originated with partial array initializer list like `double a[3]={1.0,2.0};`

AdolC-ify ISSM (3)

- so far the triumph of the "mature" tool
- but then we expanded test cases and ...
- first problem - wrong forward values computed:
  - originated with partial array initializer list like `double a[3]={1.0,2.0};`
  - forces initialization in `adouble` default constructor

AdolC-ify ISSM (3)

- so far the triumph of the "mature" tool
- but then we expanded test cases and ...
- first problem - wrong forward values computed:
  - originated with partial array initializer list like `double a[3]={1.0,2.0};`
  - forces initialization in `adouble` default constructor
  - can be disabled in Adol-C configuration

AdolC-ify ISSM (3)

- so far the triumph of the "mature" tool
- but then we expanded test cases and ...
- first problem - wrong forward values computed:
  - originated with partial array initializer list like `double a[3]={1.0,2.0};`
  - forces initialization in `adouble` default constructor
  - can be disabled in Adol-C configuration
  - also changed in Rapsodia

AdolC-ify ISSM (3)

- so far the triumph of the "mature" tool
- but then we expanded test cases and ...
- first problem - wrong forward values computed:
  - originated with partial array initializer list like `double a[3]={1.0,2.0};`
  - forces initialization in `adouble` default constructor
  - can be disabled in Adol-C configuration
  - also changed in Rapsodia
- second problem - wrong forward values computed:

AdolC-ify ISSM (3)

- so far the triumph of the "mature" tool
- but then we expanded test cases and ...
- first problem - wrong forward values computed:
  - originated with partial array initializer list like `double a[3]={1.0,2.0};`
  - forces initialization in `adouble` default constructor
  - can be disabled in Adol-C configuration
  - also changed in Rapsodia
- second problem - wrong forward values computed:
  - originated with more recent location management

AdolC-ify ISSM (3)

- so far the triumph of the "mature" tool
- but then we expanded test cases and ...
- first problem - wrong forward values computed:
  - originated with partial array initializer list like `double a[3]={1.0,2.0};`
  - forces initialization in `adouble` default constructor
  - can be disabled in Adol-C configuration
  - also changed in Rapsodia
- second problem - wrong forward values computed:
  - originated with more recent location management
  - set of fixes (partially by my, partially by K. Kulshreshta - Paderborn)

AdolC-ify ISSM (3)

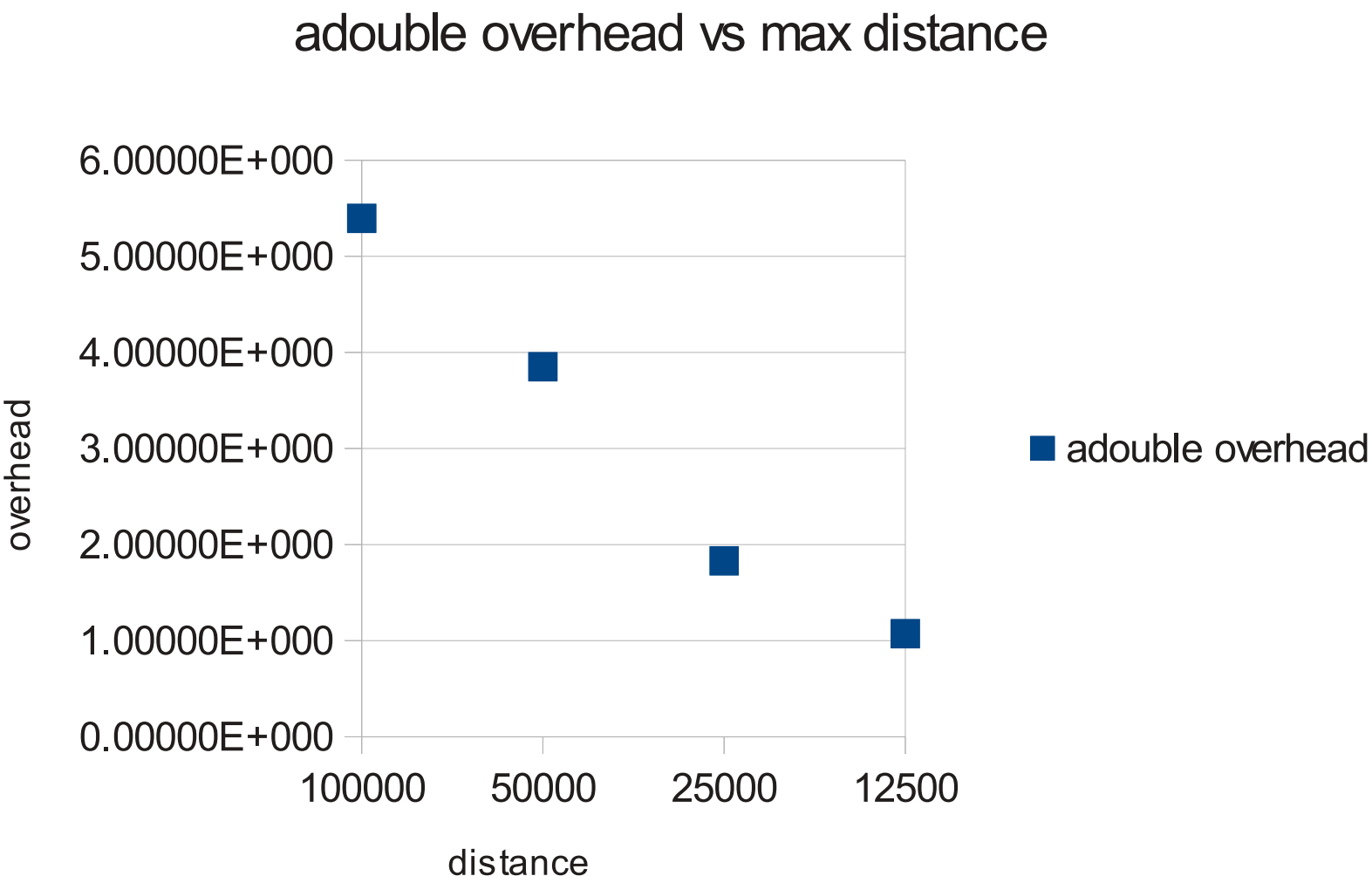
- so far the triumph of the "mature" tool
- but then we expanded test cases and ...
- first problem - wrong forward values computed:
  - originated with partial array initializer list like `double a[3]={1.0,2.0};`
  - forces initialization in `adouble` default constructor
  - can be disabled in Adol-C configuration
  - also changed in Rapsodia
- second problem - wrong forward values computed:
  - originated with more recent location management
  - set of fixes (partially by my, partially by K. Kulshreshta - Paderborn)
- overhead?

AdolC-ified ISSM performance - overloading (1)

pick some test case (here "test109"), g++ -O2, initially horrible timings fixed by another changeset to locations mgmt. from K. Kulshreshta

AdolC-ified ISSM performance - overloading (1)

pick some test case (here "test109"), g++ -O2, initially horrible timings fixed by another changeset to locations mgmt. from K. Kulshreshta

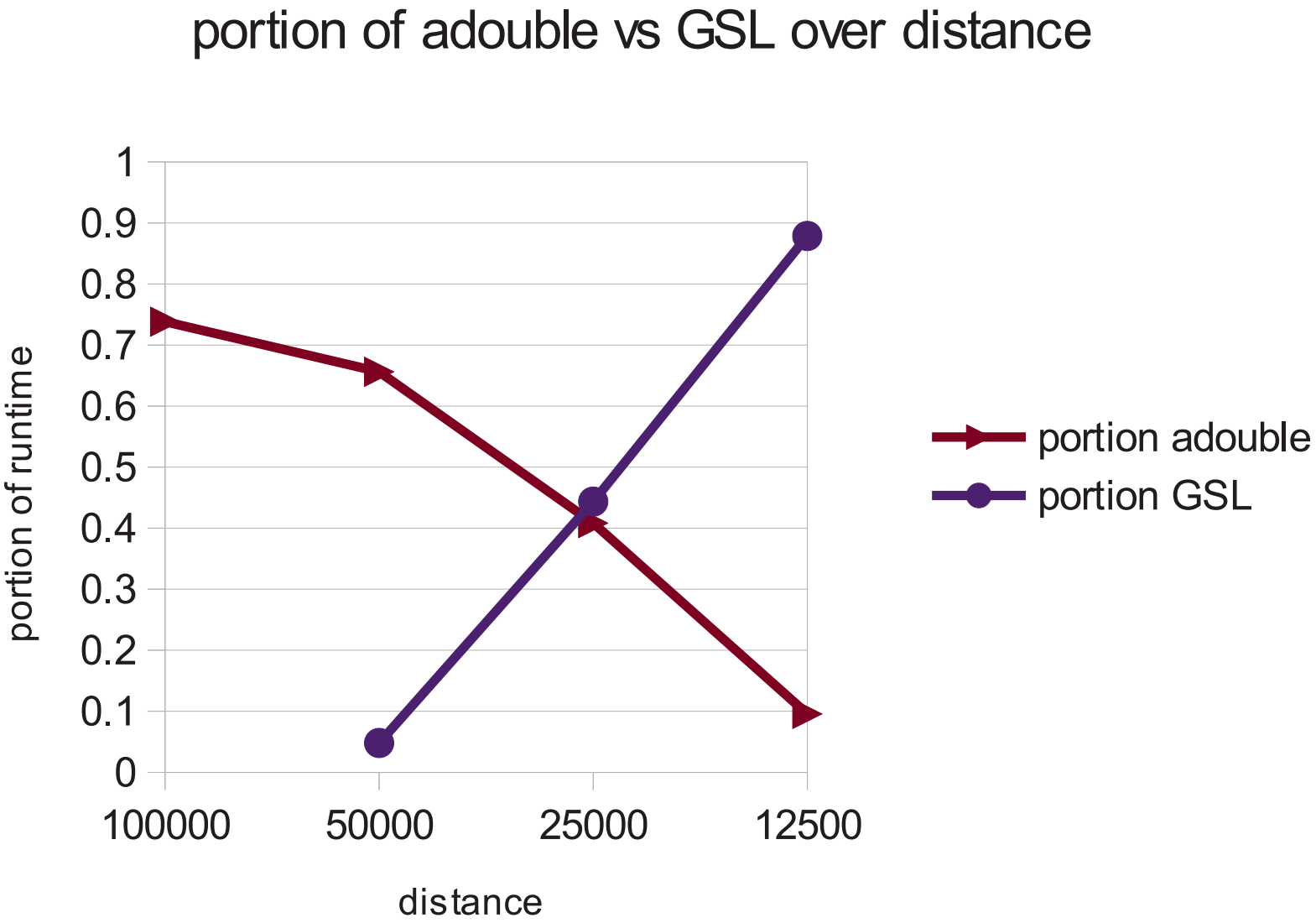


AdolC-ified ISSM performance - overloading (2)

less of a surprise once we look at the portions of runtime

AdolC-ified ISSM performance - overloading (2)

less of a surprise once we look at the portions of runtime

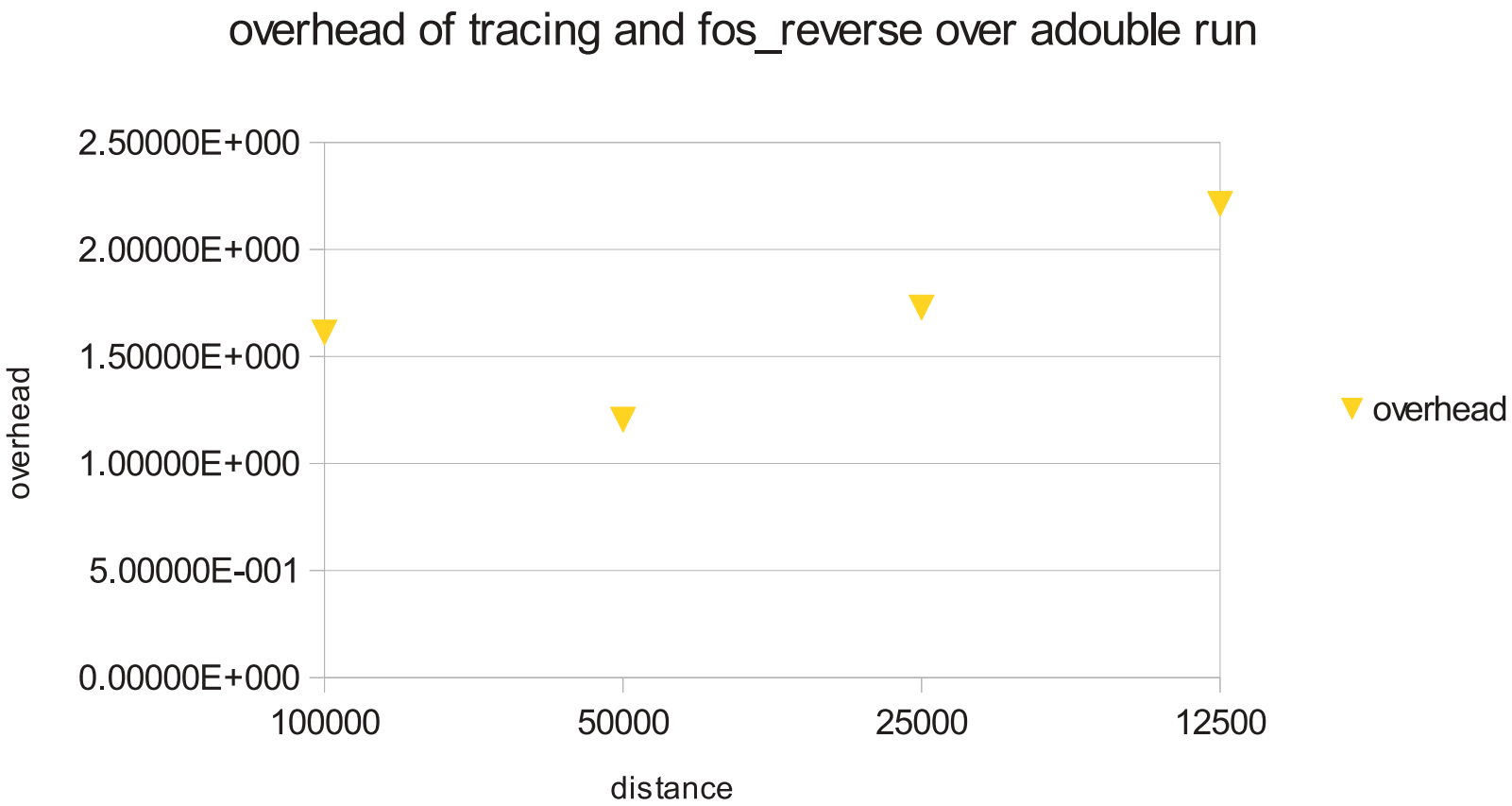


AdolC-ified ISSM performance - tracing & reverse (1)

using "test3019" - an AD-enabled regression test

AdolC-ified ISSM performance - tracing & reverse (1)

using "test3019" - an AD-enabled regression test

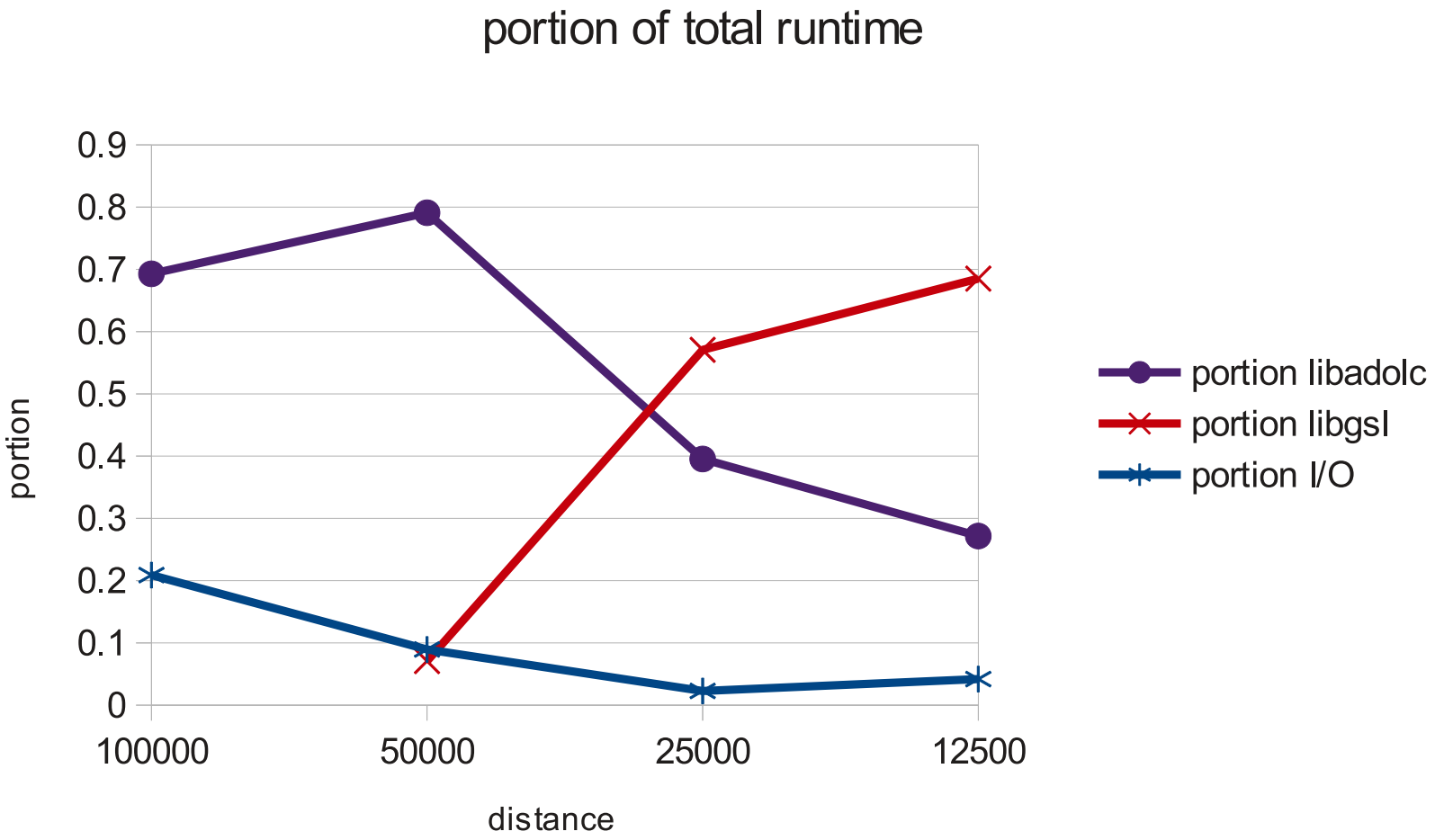


AdolC-ified ISSM performance - tracing & reverse (2)

using "test3019" - an AD-enabled regression test

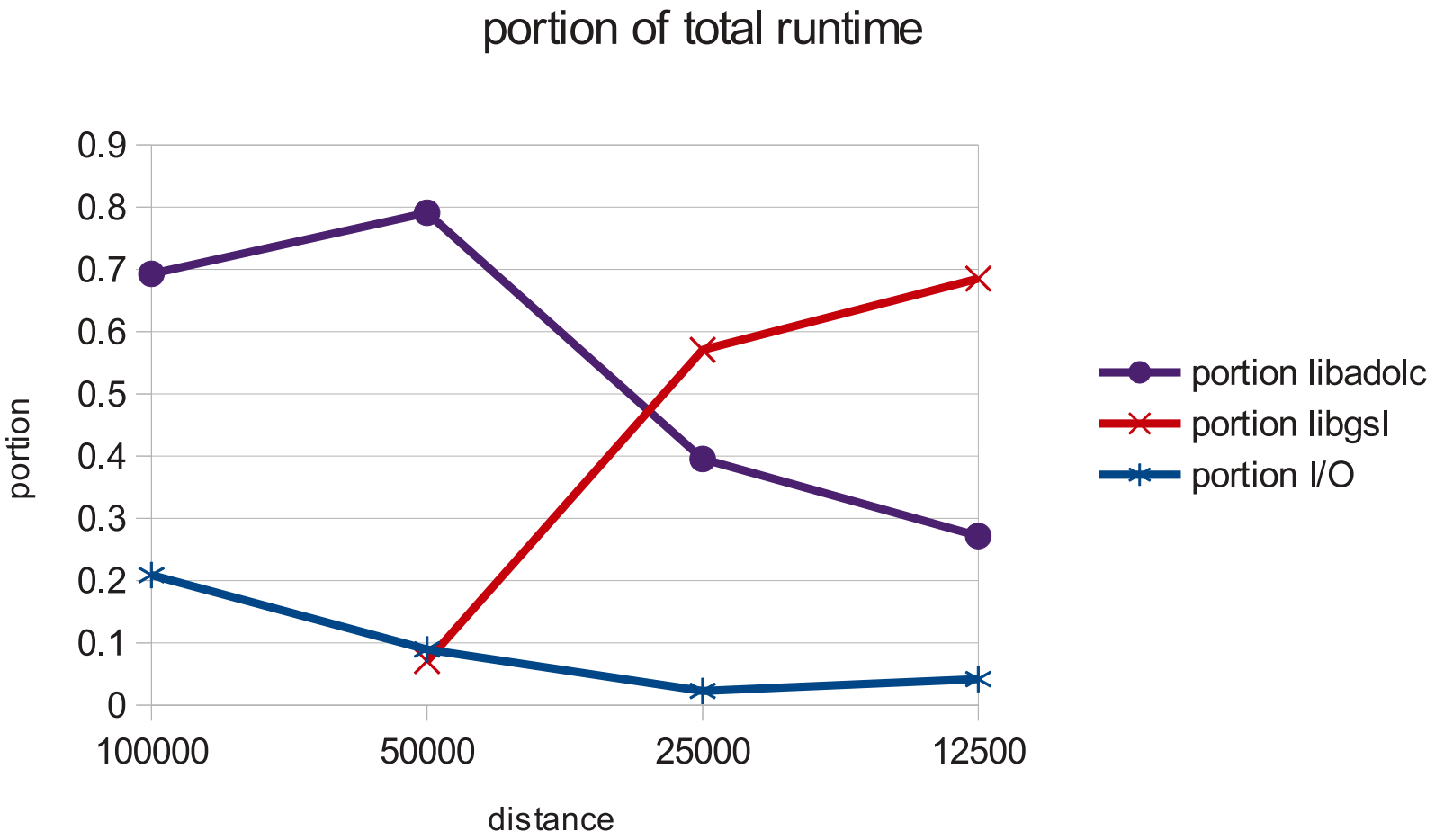
AdolC-ified ISSM performance - tracing & reverse (2)

using "test3019" - an AD-enabled regression test



AdolC-ified ISSM performance - tracing & reverse (2)

using "test3019" - an AD-enabled regression test



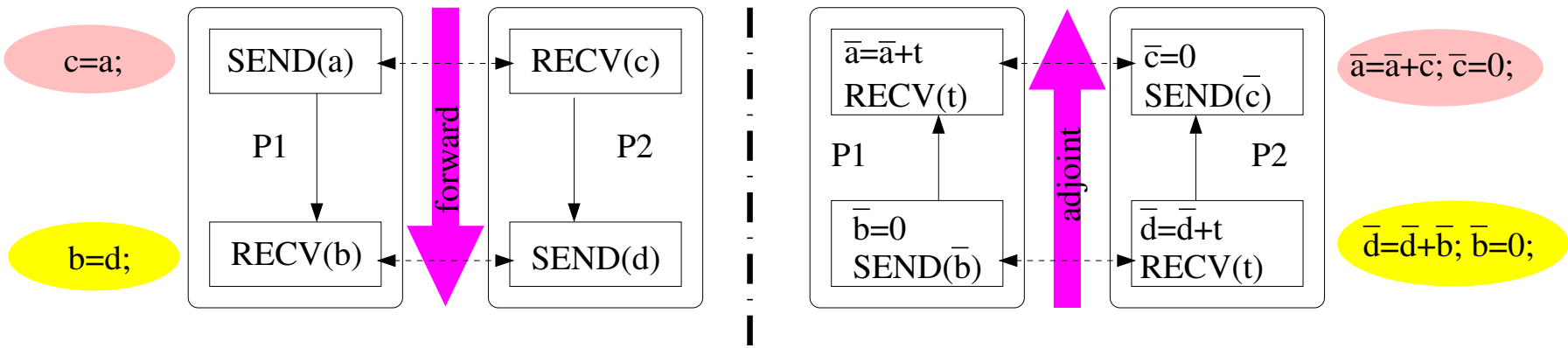
heavily skewed in Adol-C's advantage because of GSL

sidebar: Parallel with MPI (I)

- a simple MPI (pseudo) program with 6 calls :

```
mpi_init      // initialize the environment
mpi_comm_size // number of processes in the communicator
mpi_comm_rank // rank of this process in the communicator
mpi_send      // send (blocking)
mpi_recv      // receive (blocking)
mpi_finalize  // cleanup
```

- example reverse mode for blocking communication between 2 ranks and interpret as assignments



- use the communication graph as model

departing from simple setup

- different solver - MUMPS

departing from simple setup

- different solver - MUMPS
- sparse system

departing from simple setup

- different solver - MUMPS
- sparse system
- parallel setup

departing from simple setup

- different solver - MUMPS
- sparse system
- parallel setup
- had been working on Adjoinable MPI lib

  (w L. Hascoët & M. Schanen)

departing from simple setup

- different solver - MUMPS
- sparse system
- parallel setup
- had been working on Adjoinable MPI lib

  (w L. Hascoët & M. Schanen)
- most of the communication is collective which makes the adjoint "easier"; still needs the AMPI variant called

departing from simple setup

- different solver - MUMPS
- sparse system
- parallel setup
- had been working on Adjoinable MPI lib
  (w L. Hascoët & M. Schanen)
- most of the communication is collective which makes the adjoint "easier"; still needs the AMPI variant called
- don't like more preprocessor switches for using MPI or AMPI etc. littering the code

departing from simple setup

- different solver - MUMPS
- sparse system
- parallel setup
- had been working on Adjoinable MPI lib
  (w L. Hascoët & M. Schanen)
- most of the communication is collective which makes the adjoint "easier"; still needs the AMPI variant called
- don't like more preprocessor switches for using MPI or AMPI etc. littering the code
- introduce `ISSM_MPI` layer to encapsulate 4 versions

MPI	.	✓	.	✓
AD	.	.	✓	✓

departing from simple setup

- different solver - MUMPS
- sparse system
- parallel setup
- had been working on Adjoinable MPI lib
  (w L. Hascoët & M. Schanen)
- most of the communication is collective which makes the adjoint "easier"; still needs the AMPI variant called
- don't like more preprocessor switches for using MPI or AMPI etc. littering the code
- introduce `ISSM_MPI` layer to encapsulate 4 versions

MPI	.	✓	.	✓
AD	.	.	✓	✓

- MPI emulator uses `memcpy` or `adouble` assignments resp.

departing from simple setup

- different solver - MUMPS
- sparse system
- parallel setup
- had been working on Adjoinable MPI lib
  (w L. Hascoët & M. Schanen)
- most of the communication is collective which makes the adjoint "easier"; still needs the AMPI variant called
- don't like more preprocessor switches for using MPI or AMPI etc. littering the code
- introduce `ISSM_MPI` layer to encapsulate 4 versions

MPI	.	✓	.	✓
AD	.	.	✓	✓

- MPI emulator uses `memcpy` or `adouble` assignments resp.
- layer encapsulates all MPI switching $\Longrightarrow$ cleaner code

Adjoinable MPI in Adol-C and ISSM (1)

- Adol-C has:
  - (i) "tape"-based forward drivers
  - (ii) typed, 4-way stack (tape+Taylors)

Adjoinable MPI in Adol-C and ISSM (1)

- Adol-C has:
  - (i) "tape"-based forward drivers
  - (ii) typed, 4-way stack (tape+Taylors)

  makes it *"pretty outstanding in terms of uniqueness"*

  ↰ Andreas Griewank's favorite quote of a Chicago tour guide

Adjoinable MPI in Adol-C and ISSM (1)

- Adol-C has:
  - (i) "tape"-based forward drivers
  - (ii) typed, 4-way stack (tape+Taylors)

  makes it *"pretty outstanding in terms of uniqueness"*

  ↑ Andreas Griewank's favorite quote of a Chicago tour guide
- (i) means using the same AMPI internal interface as Tapenade

Adjoinable MPI in Adol-C and ISSM (1)

- Adol-C has:
  - (i) "tape"-based forward drivers
  - (ii) typed, 4-way stack (tape+Taylors)

  makes it *"pretty outstanding in terms of uniqueness"*

  ↰ Andreas Griewank's favorite quote of a Chicago tour guide
- (i) means using the same AMPI internal interface as Tapenade
- (ii) reluctant to add a 5th stack for MPI parameters with opaque types (such as communicator, datatype etc.) $\Longrightarrow$ use Adj.-MPI provided default stack

Adjoinable MPI in Adol-C and ISSM (1)

- Adol-C has:
  - (i) "tape"-based forward drivers
  - (ii) typed, 4-way stack (tape+Taylors)

  makes it *"pretty outstanding in terms of uniqueness"*
  ↑ Andreas Griewank's favorite quote of a Chicago tour guide
- (i) means using the same AMPI internal interface as Tapenade
- (ii) reluctant to add a 5th stack for MPI parameters with opaque types (such as communicator, datatype etc.) $\Longrightarrow$ use Adj.-MPI provided default stack
- problem with the above is loss of self-containedness of the trace.

Adjoinable MPI in Adol-C and ISSM (1)

- Adol-C has:
  - (i) "tape"-based forward drivers
  - (ii) typed, 4-way stack (tape+Taylors)

  makes it *"pretty outstanding in terms of uniqueness"*
  ↑ Andreas Griewank's favorite quote of a Chicago tour guide
- (i) means using the same AMPI internal interface as Tapenade
- (ii) reluctant to add a 5th stack for MPI parameters with opaque types (such as communicator, datatype etc.) $\Longrightarrow$ use Adj.-MPI provided default stack
- problem with the above is loss of self-containedness of the trace.
- general problems with (re)storing blobs are related to (de)serialization of C++ objects

Adjoinable MPI in Adol-C and ISSM (1)

- Adol-C has:
  - (i) "tape"-based forward drivers
  - (ii) typed, 4-way stack (tape+Taylors)

  makes it *"pretty outstanding in terms of uniqueness"*
  ↑ Andreas Griewank's favorite quote of a Chicago tour guide
- (i) means using the same AMPI internal interface as Tapenade
- (ii) reluctant to add a 5th stack for MPI parameters with opaque types (such as communicator, datatype etc.) $\Longrightarrow$ use Adj.-MPI provided default stack
- problem with the above is loss of self-containedness of the trace.
- general problems with (re)storing blobs are related to (de)serialization of C++ objects
- covers all interfaces needed by ISSM (reductions, gather/scatter (v) combinations)

Adjoinable MPI in Adol-C and ISSM (1)

- Adol-C has:
  - (i) "tape"-based forward drivers
  - (ii) typed, 4-way stack (tape+Taylors)

  makes it *"pretty outstanding in terms of uniqueness"*
  ↑ Andreas Griewank's favorite quote of a Chicago tour guide
- (i) means using the same AMPI internal interface as Tapenade
- (ii) reluctant to add a 5th stack for MPI parameters with opaque types (such as communicator, datatype etc.) $\Longrightarrow$ use Adj.-MPI provided default stack
- problem with the above is loss of self-containedness of the trace.
- general problems with (re)storing blobs are related to (de)serialization of C++ objects
- covers all interfaces needed by ISSM (reductions, gather/scatter (v) combinations)
- deals with `MPI_INPLACE` and 0-count buffers

Adjoinable MPI in Adol-C and ISSM (1)

- Adol-C has:
  - (i) "tape"-based forward drivers
  - (ii) typed, 4-way stack (tape+Taylors)

  makes it *"pretty outstanding in terms of uniqueness"*
  ↑ Andreas Griewank's favorite quote of a Chicago tour guide
- (i) means using the same AMPI internal interface as Tapenade
- (ii) reluctant to add a 5th stack for MPI parameters with opaque types (such as communicator, datatype etc.) $\Longrightarrow$ use Adj.-MPI provided default stack
- problem with the above is loss of self-containedness of the trace.
- general problems with (re)storing blobs are related to (de)serialization of C++ objects
- covers all interfaces needed by ISSM (reductions, gather/scatter (v) combinations)
- deals with `MPI_INPLACE` and 0-count buffers
- requires contiguous locations $\Longrightarrow$ enforced in `xNew` spec.

Adjoinable MPI in Adol-C and ISSM (2)

introduction of "active" `AMPI_ADOUBLE` type to AMPI has been thoroughly discussed between the involved parties

Adjoinable MPI in Adol-C and ISSM (2)

introduction of "active" `AMPI_ADOUBLE` type to AMPI has been thoroughly discussed between the involved parties

- have coexisting active and passive communications

Adjoinable MPI in Adol-C and ISSM (2)

introduction of "active" `AMPI_ADOUBLE` type to AMPI has been thoroughly discussed between the involved parties

- have coexisting active and passive communications
- buffers passed by `void*`

Adjoinable MPI in Adol-C and ISSM (2)

introduction of "active" `AMPI_ADOUBLE` type to AMPI has been thoroughly discussed between the involved parties

- have coexisting active and passive communications
- buffers passed by `void*`
- forces some external distinction of activity

Adjoinable MPI in Adol-C and ISSM (2)

introduction of "active" `AMPI_ADOUBLE` type to AMPI has been thoroughly discussed between the involved parties

- have coexisting active and passive communications
- buffers passed by `void*`
- forces some external distinction of activity
- most natural by corresponding MPI types, particularly when one thinks of derived MPI types

Adjoinable MPI in Adol-C and ISSM (2)

introduction of "active" `AMPI_ADOUBLE` type to AMPI has been thoroughly discussed between the involved parties

- have coexisting active and passive communications
- buffers passed by `void*`
- forces some external distinction of activity
- most natural by corresponding MPI types, particularly when one thinks of derived MPI types
- the ISSM wrapper introduces `ISSM_MPI_DOUBLE` and `ISSM_MPI_PDOUBLE`

Adjoinable MPI in Adol-C and ISSM (2)

introduction of "active" `AMPI_ADOUBLE` type to AMPI has been thoroughly discussed between the involved parties

- have coexisting active and passive communications
- buffers passed by `void*`
- forces some external distinction of activity
- most natural by corresponding MPI types, particularly when one thinks of derived MPI types
- the ISSM wrapper introduces `ISSM_MPI_DOUBLE` and `ISSM_MPI_PDOUBLE`
- correspondence required the same way as in standard MPI

Adjoinable MPI in Adol-C and ISSM (2)

introduction of "active" `AMPI_ADOUBLE` type to AMPI has been thoroughly discussed between the involved parties

- have coexisting active and passive communications
- buffers passed by `void*`
- forces some external distinction of activity
- most natural by corresponding MPI types, particularly when one thinks of derived MPI types
- the ISSM wrapper introduces `ISSM_MPI_DOUBLE` and `ISSM_MPI_PDOUBLE`
- correspondence required the same way as in standard MPI
- practical problems with collectives distributed in the code proved it is easy to get wrong

Adjoinable MPI in Adol-C and ISSM (2)

template the MPI logic with 2 parameters like this pattern

```
1  #include <iostream>
2  typedef int DataType;
3  class TypeInfo {
4  public:
5    static DataType ourDoubleType;
6    static DataType ourIntType;
7  };
8  DataType TypeInfo::ourDoubleType;
9  DataType TypeInfo::ourIntType;
10
11 template <class T, DataType *typeOfT_p> class C {
12 public:
13   C(){};
14   ~C(){};
15   void foo(T aT) { std::cout << aT << " of type " << *typeOfT_p << std::endl; }
16 };
17
18 int main (void) {
19   TypeInfo::ourDoubleType=1;
20   TypeInfo::ourIntType=2;
21   C<double,&TypeInfo::ourDoubleType>().foo(2.0);
22   C<int,&TypeInfo::ourIntType>().foo(-1);
23   return 0;
24 }
```

not completed (yet) in ISSM

wrapping MUMPS

- needs to convey sparsity info to wrapped solver

wrapping MUMPS

- needs to convey sparsity info to wrapped solver
- added integer array parameter to a second set of external func interfaces

wrapping MUMPS

- needs to convey sparsity info to wrapped solver
- added integer array parameter to a second set of external func interfaces
- suggests `void*` blobs again but has to address the same concerns as MPI opaque parameters

wrapping MUMPS

- needs to convey sparsity info to wrapped solver
- added integer array parameter to a second set of external func interfaces
- suggests `void*` blobs again but has to address the same concerns as MPI opaque parameters
- question whether to factorize again in the reverse sweep or recover factors:
  - generally - tradeoff fill-in for refactoring

wrapping MUMPS

- needs to convey sparsity info to wrapped solver
- added integer array parameter to a second set of external func interfaces
- suggests `void*` blobs again but has to address the same concerns as MPI opaque parameters
- question whether to factorize again in the reverse sweep or recover factors:
  - generally - tradeoff fill-in for refactoring
  - specifically - MUMPS can dump factors but is written in / geared toward Fortran

wrapping MUMPS

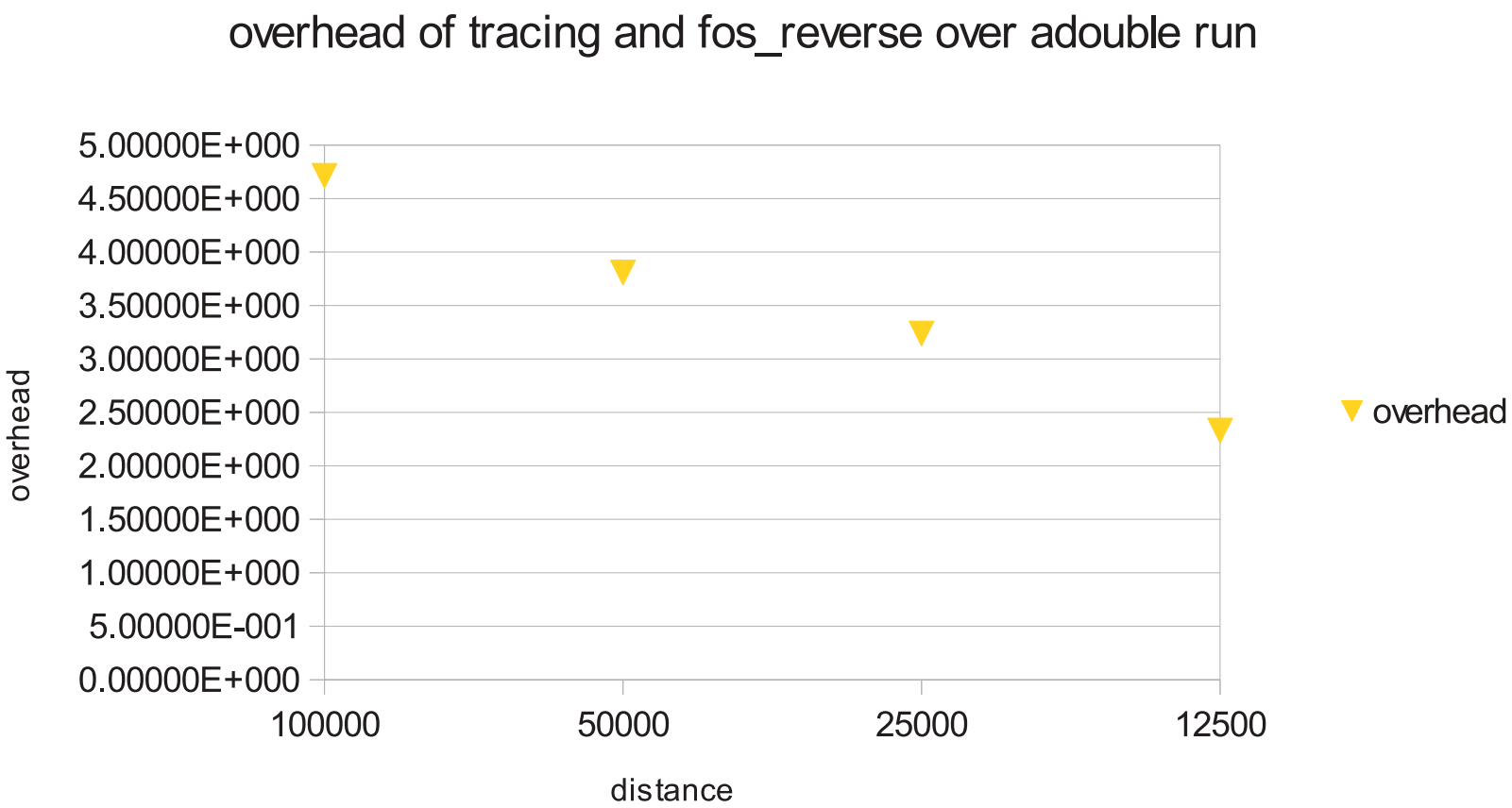
- needs to convey sparsity info to wrapped solver
- added integer array parameter to a second set of external func interfaces
- suggests `void*` blobs again but has to address the same concerns as MPI opaque parameters
- question whether to factorize again in the reverse sweep or recover factors:
  - generally - tradeoff fill-in for refactoring
  - specifically - MUMPS can dump factors but is written in / geared toward Fortran
  - performance analysis shows it is fast anyway (unlike GSL)

wrapping MUMPS

- needs to convey sparsity info to wrapped solver
- added integer array parameter to a second set of external func interfaces
- suggests `void*` blobs again but has to address the same concerns as MPI opaque parameters
- question whether to factorize again in the reverse sweep or recover factors:
  - generally - tradeoff fill-in for refactoring
  - specifically - MUMPS can dump factors but is written in / geared toward Fortran
  - performance analysis shows it is fast anyway (unlike GSL)
- revisit performance (by now $>$ 500 svn changesets later)

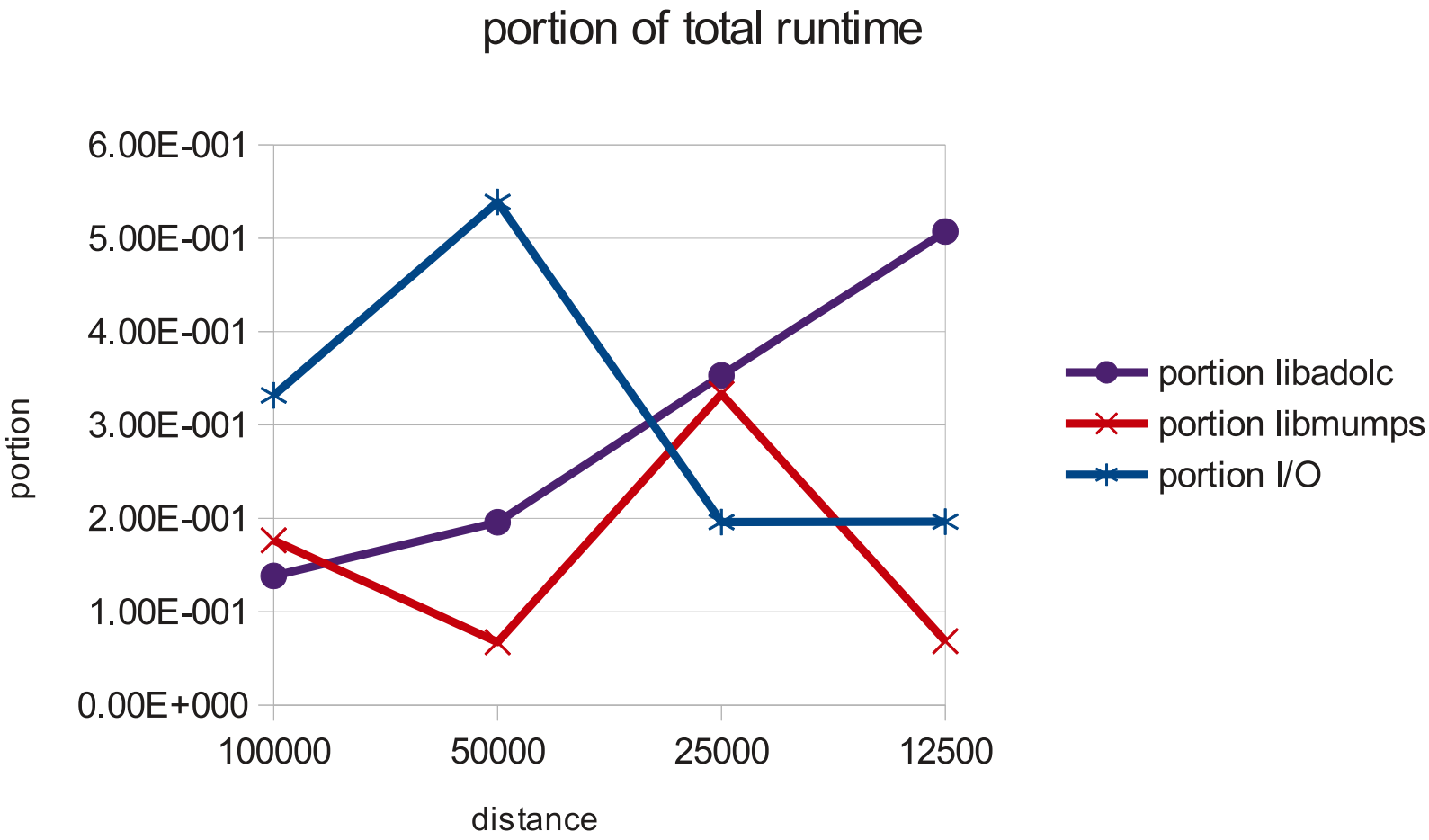
AdolC-ified ISSM performance - tracing & reverse (1)

test3019 - with contiguous locations, 3-way parallel MUMPS



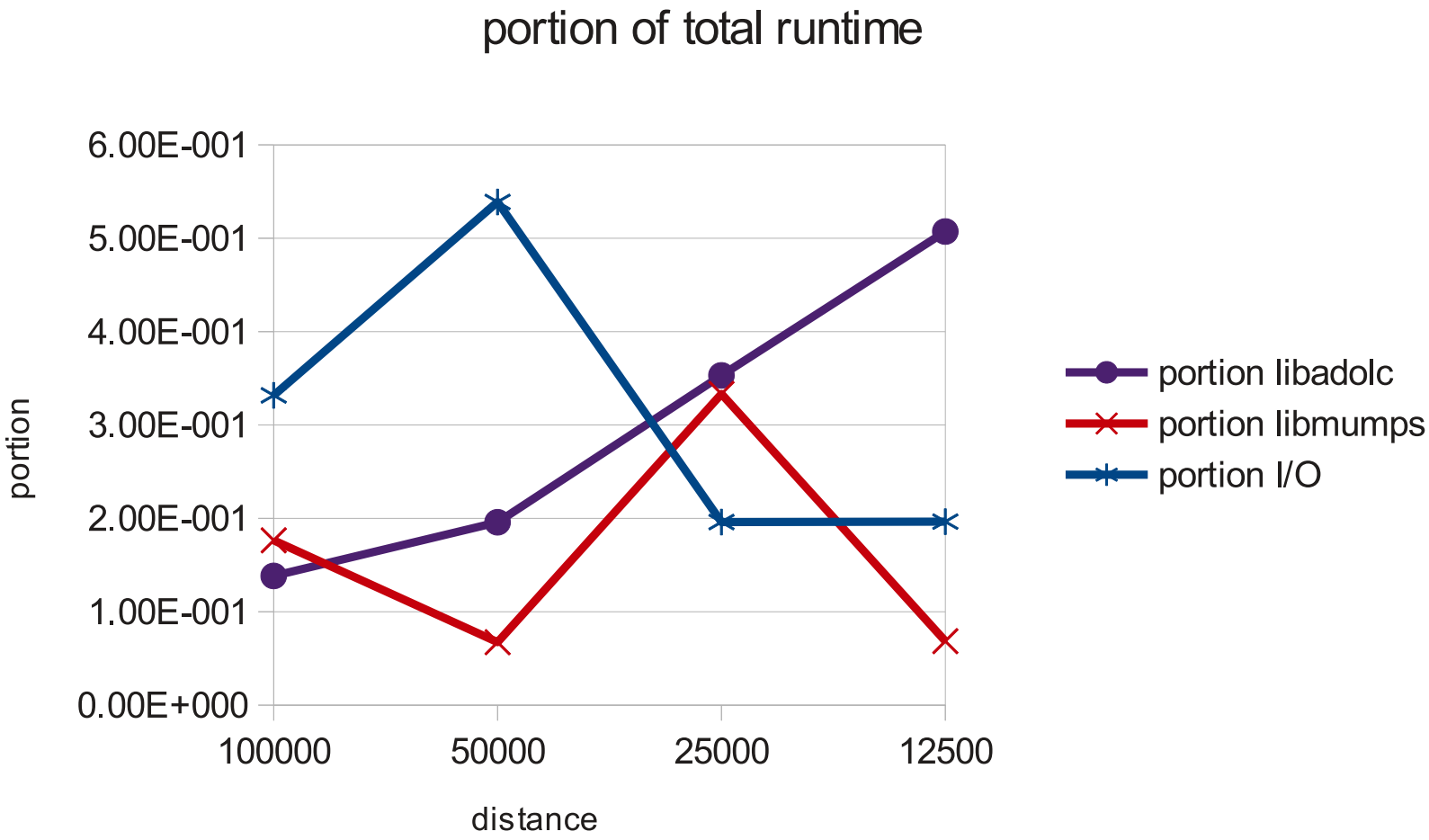
AdolC-ified ISSM performance - tracing & reverse (2)

test3019 - with contiguous locations, 3-way parallel MUMPS



AdolC-ified ISSM performance - tracing & reverse (2)

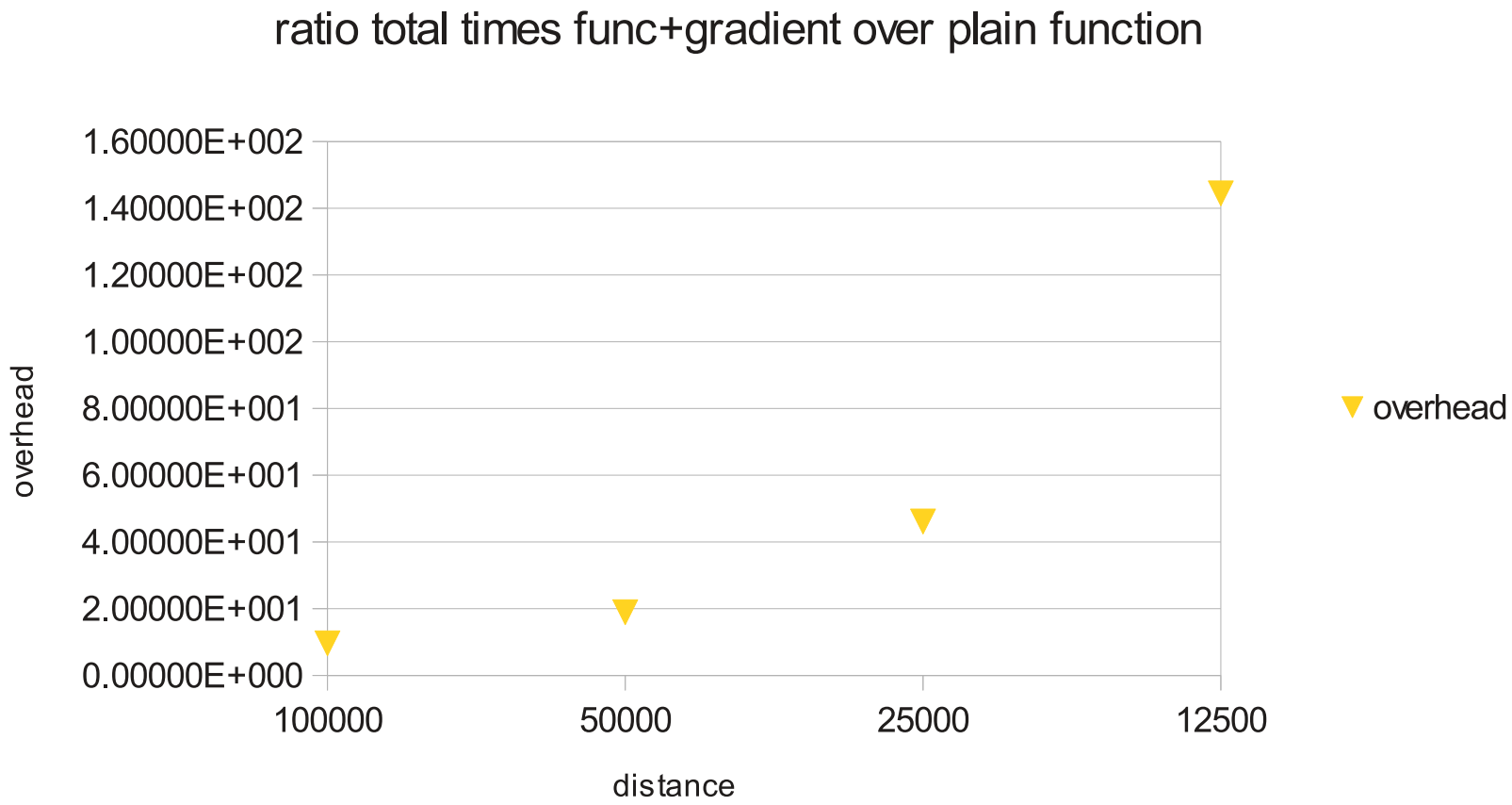
test3019 - with contiguous locations, 3-way parallel MUMPS



MUMPS is fast - more realistic picture

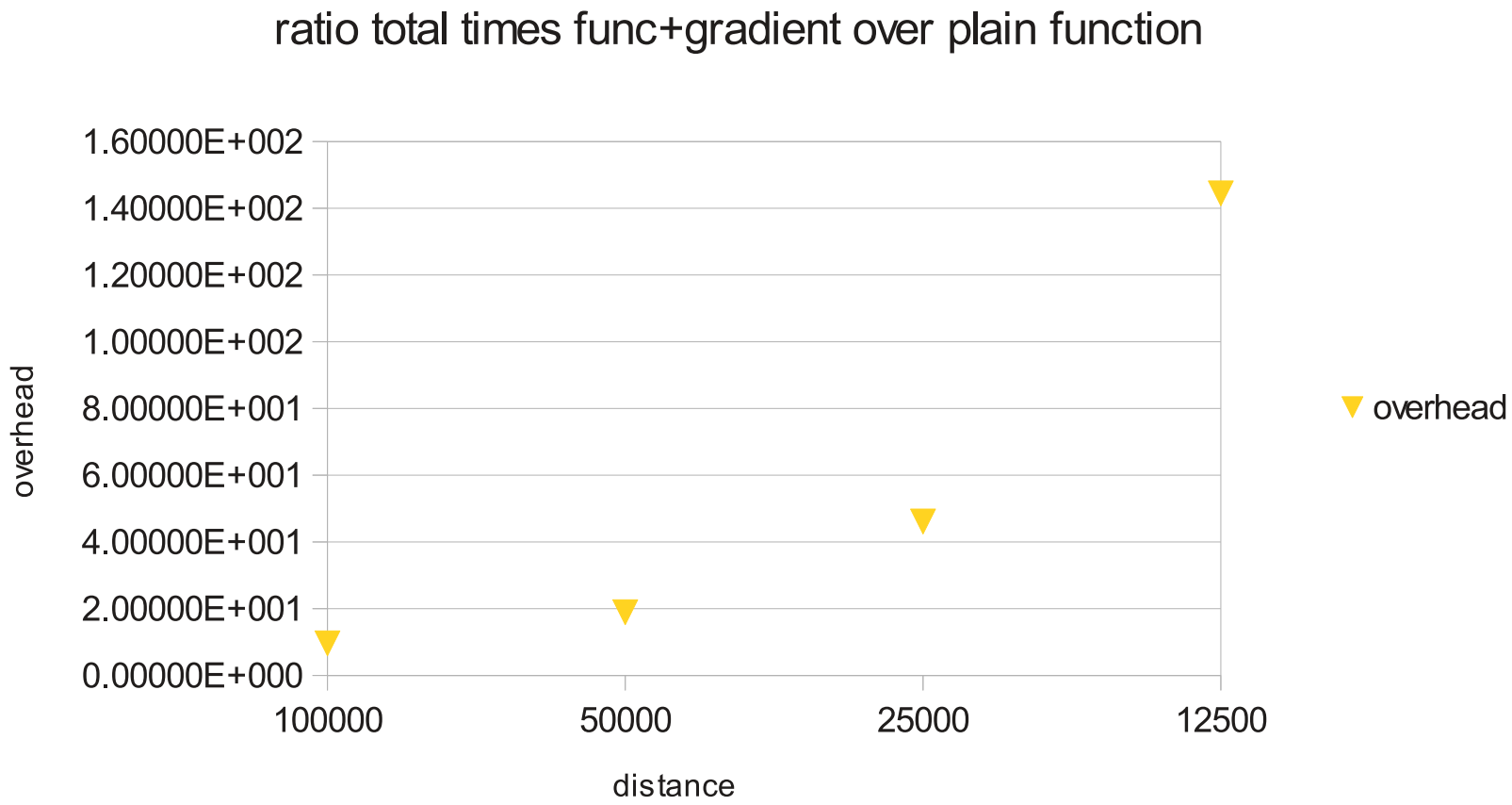
AdolC-ified ISSM performance - tracing & reverse (3)

test3019 - with contiguous locations, 3-way parallel MUMPS



AdolC-ified ISSM performance - tracing & reverse (3)

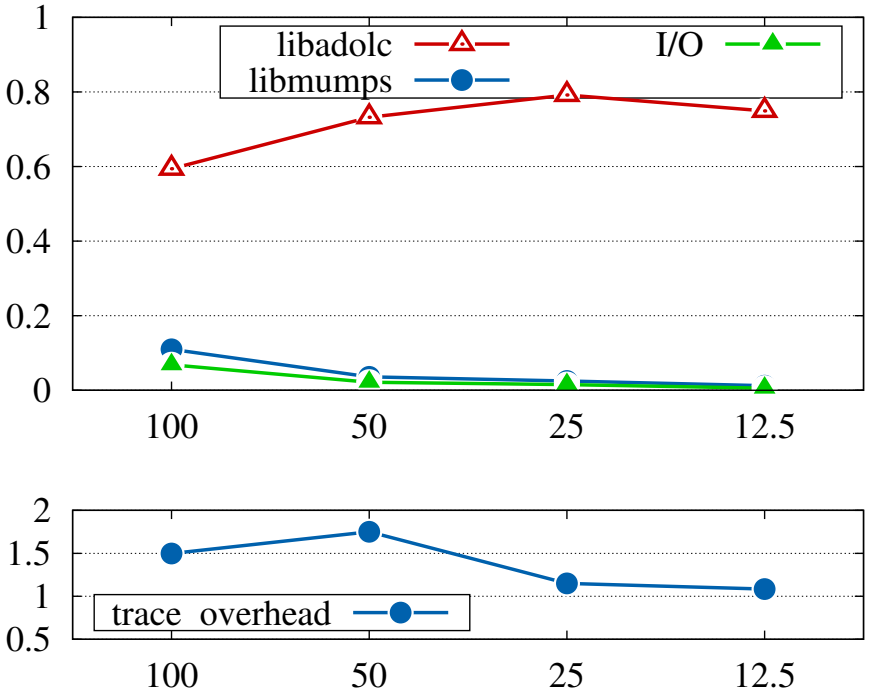
test3019 - with contiguous locations, 3-way parallel MUMPS



i.e. pretty nasty ... BUT

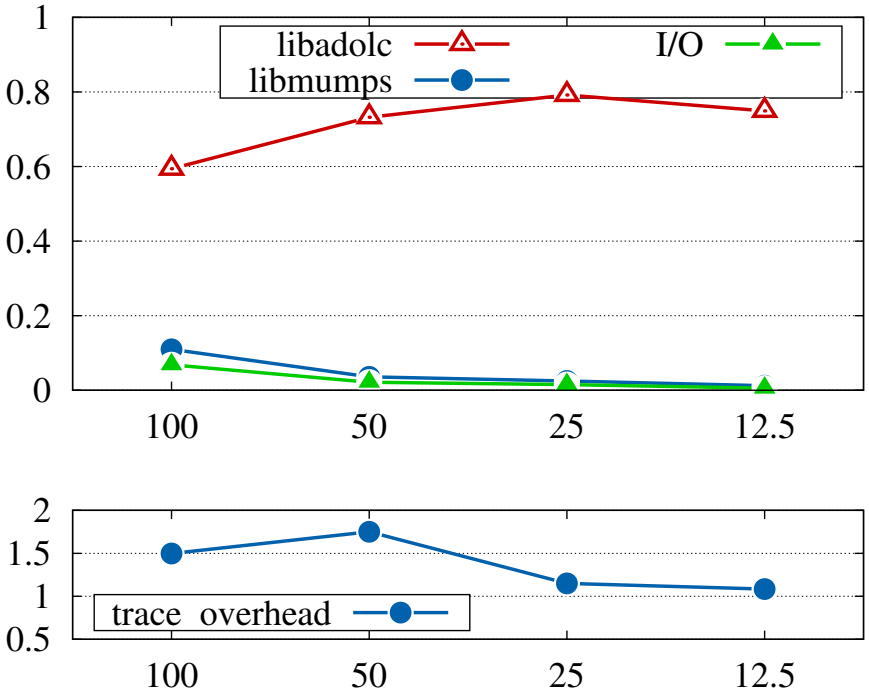
AdolC-ified ISSM performance - tracing & reverse (3)

new 3-way parallel MUMPS



AdolC-ified ISSM performance - tracing & reverse (3)

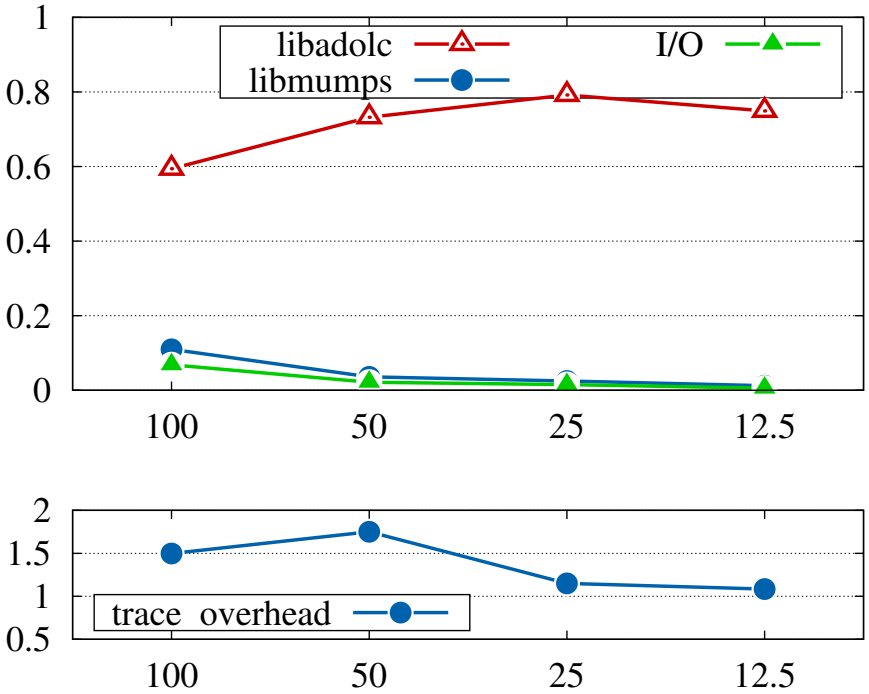
new 3-way parallel MUMPS



- realistic runtime overhead factors between 10 and 30

AdolC-ified ISSM performance - tracing & reverse (3)

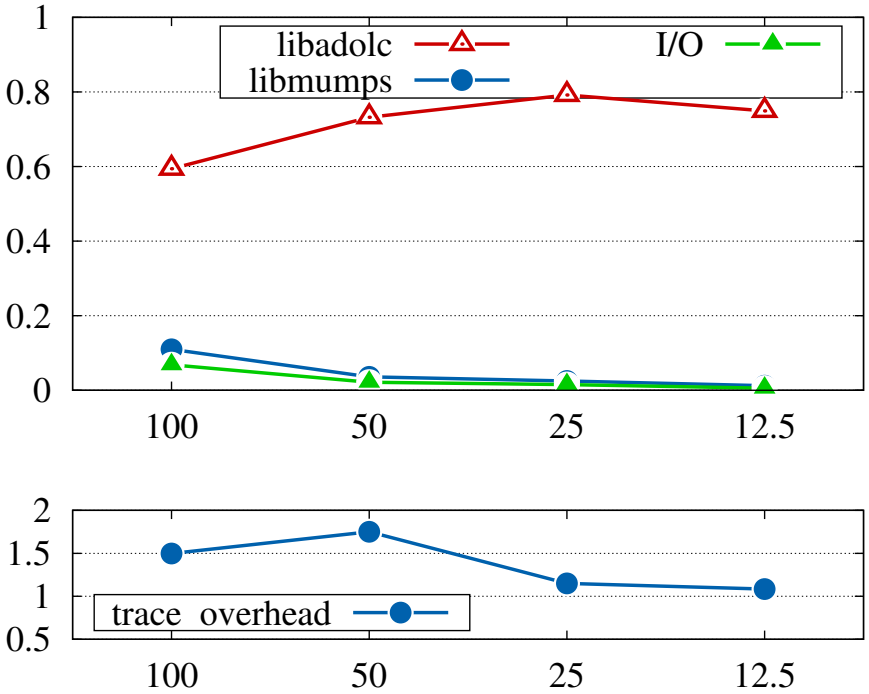
new 3-way parallel MUMPS



- realistic runtime overhead factors between 10 and 30
- reflects theoretical result for reverse interpretation

AdolC-ified ISSM performance - tracing & reverse (3)

new 3-way parallel MUMPS



- realistic runtime overhead factors between 10 and 30
- reflects theoretical result for reverse interpretation
- practically viable on Pleiades

AdolC-ified ISSM performance - outlook

since these tests happened

- 60 - way runs on Pleiades

AdolC-ified ISSM performance - outlook

since these tests happened

- 60 - way runs on Pleiades
- transient runs tax the file system and sometimes that causes crashes

AdolC-ified ISSM performance - outlook

since these tests happened

- 60 - way runs on Pleiades
- transient runs tax the file system and sometimes that causes crashes
- acc. to Eric Larour: **"You have to talk to it with love, and then you get numbers"**

AdolC-ified ISSM performance - outlook

since these tests happened

- 60 - way runs on Pleiades
- transient runs tax the file system and sometimes that causes crashes
- acc. to Eric Larour: **"You have to talk to it with love, and then you get numbers"**
- resilience/adjoint integrated checkpointing isn't there yet, but is in the works

AdolC-ified ISSM performance - outlook

since these tests happened

- 60 - way runs on Pleiades
- transient runs tax the file system and sometimes that causes crashes
- acc. to Eric Larour: **"You have to talk to it with love, and then you get numbers"**
- resilience/adjoint integrated checkpointing isn't there yet, but is in the works **IOW ... to be continued ...**